UNIVERSITY OF OSLO
Department of Informatics

Specification and Analysis of Priced Systems in Priced-Timed Maude

Master thesis

Leon Bendiksen

February 1, 2008

Abstract

This thesis investigates the suitability of extending the rewriting-logic-based Maude framework, in particular Real-Time Maude, to support the formal modeling and analysis of untimed and timed *priced* systems. The first contribution of this thesis is to define *priced* and *priced-timed rewrite theories*, show the soundness of these definitions, and prove that priced-time rewrite theories contain as a proper subset the set of *priced-timed automata* (PTA). Since all priced systems that I have encountered have been real-time systems, I focus on priced real-time (priced-timed) systems. The second main contribution of the thesis is the development of a tool, Priced-Timed Maude, supporting the specification and analysis of useful subclasses of priced and priced-timed rewrite theories. In particular, Priced-Timed Maude supports the specification of the large and important class of "flat" *object-oriented* priced-timed systems, for which I have developed useful and intuitive specification techniques. This thesis then applies Priced-Timed Maude to three larger systems, two of which can be considered benchmarks for priced-timed systems and are often encountered in the literature, and one which has been inspired by a "regular" problem found in optimization literature. I have also modeled and analyzed one of these systems using the only well known formal tool for priced-timed systems that I have found, the PTA tool UPPAAL CORA, and have compared the performance of these Priced-Timed Maude and UPPAAL CORA specifications. Unsurprisingly, UPPAAL CORA outperforms Priced-Timed Maude when analyzing this problem. This is natural, since the PTA model is quite restrictive. On the other hand, Priced-Timed Maude is more general and expressive, and lets us model more complex systems with advanced data types and communication features in an elegant and intuitive style. Furthermore, Priced-Timed Maude supports a wide range of formal analysis methods, including: *rewriting* for simulation, *search* for reachability analysis, *linear temporal logic model checking*, and finding cost- and time-optimal solutions.

Acknowledgements

Great appreciation is extended to the following:

My supervisor Peter C. Ölveczky for applying his expertise, giving good advice, and his red and blue pens.

To Ryel Alviola for the semi-intelligible mumblings, giving helpful feedback, and for coining the phrase 'with tears in his eyes'.

To my mom and Gaia for providing me with the means to survive on a diet of pizza, pick-me-up Coke, and other healthy foodstuff.

To my good friends Roy and kaimt for moral support and assistance in consuming large amounts of pizza and pick-me-up Coke.

Contents

Chapter 1

Introduction

Real-Time Maude [1, 2] is a timed extension of the rewriting logic tool Maude [3] that has been successfully applied to a wide range of advanced applications (see, e.g., [4, 5, 6, 7, 8, 9]). Real-Time Maude distinguishes itself from other formal tools for real-time systems because of its expressiveness, generality, and support for object-oriented specification. In fact, all the advanced applications it has been applied to have been modeled as object-oriented systems. However, in some real-time systems (and in some untimed systems) *cost* is an important dimension, whether it be the cost of instantaneous transitions or the cost accumulating at a certain rate over time. Cost in a system can, for instance, be the power consumed by a processor while it is working on a certain task, or the power consumed when turning on a light bulb. One formal tool that supports the specification and analysis of priced real-time (priced-timed) systems is Uppaal CORA [10], which is a priced extension of the timed automaton tool Uppaal [11]. However, this tool does not support object-oriented specification, the specification of advanced functions and data types, etc.

This thesis investigates the suitability of using Real-Time Maude for the specification and analysis of priced-timed systems. In particular, it investigates the suitability of modeling such systems as object-oriented systems. To this end, we define a theory for modeling priced and priced-timed systems as *priced* and *priced-timed rewrite theories*. As part of the investigation, a tool, *Priced-Timed Maude*, has been developed to support the specification and analysis of priced and priced-timed rewrite theories. Specifically, Chapter 3 defines the concepts of priced and priced-timed rewrite theories and proves that these models are sound by showing that equivalent rewrite proofs have the same cost. Despite the fact that we have shown priced rewrite theories to be sound, as a result of implementation-specific issues, the current prototype of Priced-Timed Maude only supports flat and flat object-oriented systems. However, *all* the advanced problems Real-Time Maude have been applied to have been specified as flat object-oriented systems. Therefore, these classes cover large and important classes of priced-timed systems. The class of systems that can be specified using these techniques exceeds what can be specified in Uppaal CORA. Chapter 4 gives an example of a system, a priced thermostat, that cannot be specified in Uppaal CORA.

Chapter 4 introduces the Priced-Timed Maude specification language. It gives general specification techniques for object-oriented specification of priced-timed systems. The specification techniques extend those used for the advanced applications of Real-Time Maude.

Chapter 5 presents the new commands provided by Priced-Timed Maude. In addition to extending Real-Time Maude commands with support for priced-timed systems, Priced-Timed Maude also provides commands to obtain the cheapest cost for reaching a state.

In Chapter 6 the semantics of Priced-Timed Maude is defined in Real-Time Maude. Priced-Timed Maude commands are implemented by transforming the module and the command into a pair consisting of a Real-Time Maude module and Real-Time Maude command. Because Maude is reflective, Priced-Timed Maude modules and terms can be meta-represented as terms in Maude, and can be manipulated by Maude functions. Priced-Timed Maude is therefore developed in Maude as an extension of Real-Time Maude.

Chapter 7 presents three larger case studies:

- The first problem is the *airplane landing problem* (ALP) [12]. In this problem, aircraft landings are scheduled within a given time window onto a set of runways. If an aircraft is assigned a landing time that deviates from a given target time, then it has to accelerate or hold in the air. This leads to more fuel consumption than planned, and an additional cost is incurred. Therefore, the objective is to minimize this cost. This problem was chosen because ALP is often cited in papers discussing priced-timed systems (e.g., [13, 14]).
- The second problem is *energy task graph scheduling* (ETGS) [13]. In this problem, we are given a set of interdependant tasks, a set of processors, and a bus. The tasks require a given time to run on each processor and to broadcast on the bus. The processors and bus consume power at a given rate while processing/broadcasting and a different rate of power while idle. The objective is to find the most energy-efficient schedule for completing all the tasks within a given deadline. This example was chosen because of its practical applications and is the one cited in [13]. It is also used in Chapter 8 when comparing Priced-Timed Maude's version of the specification to Uppaal CORA's.
- The last problem is a *subway passenger routing* (SPR) problem. This problem deals with routing passengers who want to travel within a subway network. Each train uses a set amount of power based on how many cars are attached to it. The objective is to minimize the trains' total power consumption, while at the same time making sure all passengers reach their destinations. This problem is presented mainly to illustrate how more complex systems, that seem hard or impossible to model using traditional automata-based tools for analyzing priced-timed systems, can be modeled and analyzed in Priced-Timed Maude.

These examples are all implemented as object-oriented Priced-Timed Maude specifications.

Chapter 8 compares priced-timed automata with priced-timed rewrite theories and shows that priced-timed rewrite theories are more expressive, since any priced-timed automaton can be expressed as a priced-timed rewrite theory, while the converse is not true. Thereafter, I model the ETGS example in Uppaal CORA, and run a performance comparison between the two. Not surprisingly, Uppaal CORA outperforms Priced-Timed Maude by a great margin as this is an example that can easily be modeled in Uppaal CORA, which is fine-tuned for this kind of problem.

Finally, Chapter 9 gives some concluding remarks.

Chapter 2

Rewriting Logic, Maude, and Real-Time Maude

This chapter presents some background on *rewriting logic* [15], Maude [3], and Real-Time Maude [1, 2]. Rewriting logic uses *membership equational logic* [16] to model the static parts of a system and *rewrite rules* to model its dynamic parts. Maude is a high performance tool supporting the specification and analysis of rewriting logic theories. Maude has been shown to be an intuitive and powerful tool for specifying and analyzing dynamic systems [17]. In particular, it has proven itself useful in modeling distributed systems in an object-oriented style. Real-Time Maude is a timed extension of Maude that has been successfully applied to a wide range of advanced applications (see, e.g., [4, 5, 6, 7, 8, 9]). Priced-Timed Maude extends Real-Time Maude and is implemented in Maude.

2.1 Rewriting Logic

An equational specification (Ω, E) contains a signature Ω that declares function symbols, sorts, and subsorts, and a set of equations E that define the non-constructor functions in Ω. In this thesis, we use only order-sorted equational specification [18], where the signature is $(S, \leq, \Sigma)$, with S is a set of sorts, $\leq$ the subsort operator, and Σ a set of function symbols. In order-sorted signatures, the sorts may be related to each other with the subsort operator. Intuitively, $S \leq S'$ means that S is a subsort of S', so that each term of sort S is also a term of sort S'.

A rewrite theory is a tuple $\mathcal{R} = (\Omega, E, L, R)$ where (Ω, E) is an equational specification, L a set of labels, and R is a set of conditional and unconditional rewrite rules. For conditional rewrite rules we write

$$l \ : \ t \ \longrightarrow \ t' \ \texttt{if} \ \mathit{cond}$$

where the label $l \in L$, and is understood as the term t rewritten to the term t' by the rule l and is applied only if the condition *cond* holds. Such a rule models a *local state change* from an instance of t to the corresponding instance of t'. For unconditional rewrite rules we write

$$l \ : \ t \ \longrightarrow \ t'$$

Rewrite rules define dynamic transitions in a system and may only be applied from left to right. For $\mathcal{R}$ a rewrite theory, $\mathcal{R} \vdash t \longrightarrow u$ means that u is reachable from t (in zero or more steps). This provability relation is defined as follows (from [19]):

Definition 2.1.1 (Deduction rules of rewriting logic) *Given a rewriting logic specification $\mathcal{R} = (\Omega, E, L, R)$ (which we for simplicity assume is one-sorted and has only unconditional rules), the sequent*

$$\mathcal{R} \vdash t \longrightarrow u$$

holds if and only if $t \longrightarrow u$ can be obtained by finite application of the following rules of deduction:

Reflexivity: *For each term t in $\mathcal{T}_\Omega(X)$*

$$t \longrightarrow t$$

holds.

Equality: *If $t \longrightarrow t'$ holds, and $E \vdash t = u$ and $E \vdash t' = u'$ both hold, then*

$$u \longrightarrow u'$$

holds.

Congruence: *For each function symbol f in Ω, if $t_1 \longrightarrow u_1$, ..., and $t_n \longrightarrow u_n$ all hold, then*

$$f(t_1, \ldots, t_n) \longrightarrow f(u_1, \ldots, u_n)$$

holds.

Replacement: *For each rewrite rule $l : t(x_1, \ldots, x_n) \longrightarrow u(x_1, \ldots, x_n)$ in R, if $t_1 \longrightarrow u_1$, ..., and $t_n \longrightarrow u_n$ all hold, then*

$$t(t_1/x_1, \ldots, t_n/x_n) \longrightarrow u(u_1/x_1, \ldots, u_n/x_n)$$

holds. (here t_i/x_i means that each x_i is simultaneously replaced by t_i.)

Transitivity: *If $t_1 \longrightarrow t_2$ and $t_2 \longrightarrow t_3$ both hold, then*

$$t_1 \longrightarrow t_3$$

holds.

It is important to note that rewrite rules are modulo the equations of E according to the **Equality** rule of this definition.

2.2 Specification of Rewrite Theories in Maude

Maude [3] is a tool and specification language for membership equational logic and rewriting logic. In Maude, sorts are declared using the `sort` or `sorts` keyword using the following syntax:

```
sorts s1 ... sn .
```

For instance, `sorts Nat Int .` defines two new sorts `Nat` and `Int`. Subsorts are declared using the `subsort` keyword, which is used with the syntax:

`subsort` s_1 ... s_n `<` s_i ... s_j `.`

Variables in Maude do not represent memory locations like in programming languages such as Java; instead, they are mathematical so that equations and rules are implicitly universally quantified with regard to the variables. Each variable has a name and a sort and can either be declared using the syntax:

`var` *name* `:` *sort* `.`

or explicitly *name*`:`*sort*. For instance, `X:Nat` is a variable of sort `Nat`.

Function symbols are declared using the `op` or `ops` keywords. Underscores can be used to indicate the position of arguments. Operators are defined with syntax:

`op` *opName* `:` s_1 ... s_n `->` s_r [`[`*Atts*`]`] `.`

where $s_1 \ldots s_n$ are the sorts of the arguments of the function (a function with 0 arguments is a constant); s_r is the range of the function; and '`[`*Atts*`]`' is optional and declares extra properties for the operator such as `assoc`, denoting associativity; `comm`, denoting commutativity; and `id`, specifying an identity element. All matching is performed *modulo* such attributes. For instance,

```
op zero : -> Nat .
op _+_ : Nat Nat -> Nat [assoc comm id: zero] .
op dbl : Nat -> Nat .
```

defines the constant `zero`, the associative and commutative operator `_+_` with `zero` as identify, and the operator `dbl`.

Equations are used to define functions in Maude. They can be either conditional or unconditional, and are written with syntax

`ceq` t `=` u `if` *cond* `.`

for conditional equations and

`eq` t `=` u `.`

for unconditional equations. For instance, to define the function `dbl` that doubles its argument, we could give the equation

```
eq dbl(X:Nat) = X:Nat + X:Nat .
```

(assuming that the `_+_` operator i defined). The equations in Maude should be terminating and confluent.

A conditional rewrite rule

$l : t \longrightarrow t'$ `if` *cond*

is written with syntax:

`crl [`l`] :` t `=>` t' `if` *cond* `.`

As mentioned, rewrite rules are used to rewrite terms between canonical forms. The left-hand side of all rewrite rules *must* be the canonical form of a term, as Maude automatically reduces all terms to this form before applying the rules to them.

To write comments in Maude we use `***` or `---` for one line comments, while multi-line comments are encapsulated in `***(...)` or `---(...)`.

Equational specifications are defined using the functional module type in Maude and are written with syntax:

`fmod` *MODULENAME* `is`
BODY
`endfm`

Inclusion of modules is performed by using the keyword `including` or `protecting` followed by the name of the module to import. The *BODY* of a functional module represents an equational specification and contains: a list of imported modules, a set of declared sorts, a list of subsort declarations, a set of operator declarations, a set of membership axioms, and a set of equations. Rewrite theories are defined using system modules with the syntax:

`mod` *MODULENAME* `is`
BODY
`endm`

In addition to what a functional module contains, the *BODY* of a system module may also contain a set of rewrite rules.

2.3 Object-Oriented Specification in Maude

For object-oriented specifications the module type `omod` with syntax `omod` *MODULENAME* `is` *BODY* `endom` is used. Classes are declared with the keyword `class` followed by a class name and a set of attributes:

`class` *ClassName* `|` $attribute_1$ `:` s_1, ..., $attribute_n$ `:` s_n `.`

For instance, we can declare a switch for light switches with a status and a light bulb as follows:

```
class LightSwitch | status : Status, bulb : Nat .
```

Subclasses can be declared using the keyword `subclass`. A subclass inherits all the attributes and rules of its superclass(es). The following syntax is used to declare a subclass:

`subclass` *SubClass* `<` *SuperClass* `.`

When instantiating objects, the sort `Oid` is used to assign unique identifiers to each of them. *Objects* are represented as terms of the form

`<` *O* `:` *ClassName* `|` $attribute_1$ `:` $value_1$, ..., $attribute_n$ `:` $value_n$`>`

where *O* is a term of sort `Oid`. When attributes are not affected by a rule, they may be omitted. For instance, if we have the following class

```
class TwoAtt | att1 : Nat, att2 : Nat .
```

and we want to specify a rule adding the number `5` to `att2`, we may use the following rule:

```
rl [add-five] : < O:Oid : TwoAtt | att2 : N:Nat >
                =>
                < O:Oid : TwoAtt | att2 : N:Nat + 5 > .
```

Maude also has built-in support for messages. Messages are used to pass information between objects in a configuration. Messages can be defined much in the same way as operators using the following syntax:

`msg` *msgType* `:` s_1 ... s_n `-> Msg .`

where *msgType* is the name of the type of the message and $s_1 \ldots s_n$ are the sorts of the parameters for the message.

The state of a distributed system can be regarded as a *multiset* of objects and messages. In Maude such multisets are terms of the following sort `Configuration`, where the multiset is a union defined by the juxtaposition of objects and messages (*syntax*):

```
sorts Object Msg Configuration .
subsort Object Msg < Configuration .

op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration [ctor config assoc comm id: none] .
```

2.4 Maude Analysis

Maude provides commands to analyze rewrite theories, such as *equational reduction* to compute the E-normal form of a term, *rewriting* to simulate one behavior of the system, *search* for reachability analysis, and *temporal logic model checking*. As already mentioned, Maude assumes that the underlying equational specification is confluent and terminating.

2.4.1 Reduction

The command `red` uses the syntax

`red` t `.`

and reduces the given term t to its canonical (or E-normal) form using the equations in a specification.

2.4.2 Rewriting

The rewriting commands `rew` and `frew` are used to simulate *one possible* behavior of the system by applying the rules in the specification successively to the initial state. This is useful for prototyping and simulation purposes. The rewriting commands use the following syntax:

`rew` [`[`*n*`]`] *init* `.`
`frew` [`[`*n*`]`] *init* `.`

where *init* is a term denoting the initial state. The '`[`*n*`]`' is optional and *n* gives an upper limit on how many rewrite steps to perform.

2.4.3 Search

Search uses a *breadth-first* strategy to explore all possible behaviors from a given initial state:

`search` [`[`*n*`]`] *init* `=>*` *searchPattern* [`such that` *cond*] `.`

searches for states reachable from *init* that match the pattern *searchPattern* and satisfy the optional condition *cond* on the pattern. The '`[`*n*`]`' part is optional with *n* the upper bound on the number of solutions to display. The arrow `=>*` means the command searches for states that can be reached in zero or more rewrite steps. The arrow `=>!` is used to search for states that are deadlocked, i.e., may not be rewritten further.

2.4.4 Model Checking

Maude provides a high performance *linear temporal logic* (LTL) model checker that checks whether a temporal logic formula constructed from *atomic propositions* and temporal logic operators such as `~` (negation), `/\` (conjunction), `\/` (disjunction), `<>` (eventually), `[]` (always), and `U` (until) holds for *all* behaviors from a given initial state. *Parametric* atomic propositions are defined as function symbols of the sort `Prop`:

`op` *prop* `:` s_1 ... s_n `->` `Prop` `.`

The actual proposition is defined with equations by the following syntax:

`eq` *pattern* `|=` *prop*(...) `=` *b* `.`

where *b* is a boolean. If `t |=` *prop* evaluates to `true`, then *prop* holds in the state `t`.

In Maude, we invoke the model checker by giving the command

`red modelCheck(`*init*`,`*formula*`)` `.`

The model checker returns `true` if the formula holds for all behaviors. If the formula does not hold for all behaviors, a counter-example is returned. Of course, LTL model checking only terminates if the set of states reachable from the initial state is finite.

2.5 Meta-Programming in Maude

Maude modules and terms can be meta-represented as terms. We use this fact in Chapter 6 to transform specifications and terms.

Terms in Maude are meta-represented by terms of the sort `Term`. Constants and variables are meta-represented as the sorts `Constant` and `Variable` that are subsorts of the sort `Term` and `Qid`. Quoted identifiers are strings that start with the symbol `'`, e.g., `'abc`. Constants are represented as '*name*.*sort* , while a variable *name*:*sort* is represented as '*name*:*sort*. For instance, the term `'a.MySort` is the meta-representation of the constant `a` of sort `MySort` and the term `'X:MySort` is the meta-representation of the variable `X` of sort `MySort`. Variables are always declared explicitly in this form in meta-representations of rules and equations.

The following meta-operators show how term lists and operator symbols in a term are represented on the meta-level in Maude:

```
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc ...] .
op _[_] : Qid TermList -> Term [ctor] .
```

i.e., operator symbols are represented as a quoted identifier followed by a term list. For instance, the term `f(X:MySort, Y:MySort)` is meta-represented as `'f['X:MysSort, 'Y:Mysort]`.

The operators

```
op eq_=_[_]. : Term Term AttrSet -> Equation [...] .
op ceq_=_if_[_]. : Term Term EqCondition AttrSet -> Equation [...] .
```

are used when meta-representing equations. The first term represents the left-hand side and the second term represents the right-hand side of the equation. For instance, the equation

```
eq dbl(X:Nat) = X:Nat + X:Nat .
```

is meta-represented by the term `eq 'dbl['X:Nat] = '_+_['X:Nat, 'X:Nat] [none].`

Rules are meta-represented in a very similar manner using the following 2 operators:

```
op rl_=>_[_]. : Term Term AttrSet -> Rule [...] .
op crl_=>_if_[_]. : Term Term Condition AttrSet -> Rule [...] .
```

Finally, modules are meta-represented by the two sorts `Module` and `FModule` defined as follows:

```
sorts FModule Module .
subsort FModule < Module .
op fmod_is_sorts_.____endfm : Header ImportList SortSet SubsortDeclSet OpDeclSet
                               MembAxSet EquationSet -> FModule [...] .
op mod_is_sorts_.____endm : Header ImportList SortSet SubsortDeclSet OpDeclSet
                            MembAxSet EquationSet RuleSet -> Module [...] .
```

From the above we see that modules are represented in the expected way with its name, followed by a list of imported modules, a set of sorts, a list of subsort declarations, a list of operator symbols, a set of membership axioms, and a set of equations defining functional modules, i.e., equational specifications. In addition, system modules that represent rewrite theories also have a set of rules.

All the parts of modules are represented as meta-terms and can therefore be manipulated by Maude, e.g., the rules of a module may be manipulated and changed.

2.6 Real-Time Maude

Real-Time Maude [1, 2] is a tool that extends Maude with support for specification and analysis of *real-time rewrite theories* [20]. The tool has been implemented using Maude's meta-programming and reflection capabilities. Real-Time Maude has been successfully used to analyze and specify many state-of-the-art systems such as scheduling algorithms [7], wireless sensor network protocols [4, 5], network protocols [6, 8], cryptographic protocols [21], and real-time resource-sharing protocols [9]. All these systems have been modeled in Real-Time Maude using an object-oriented style.

A Real-Time Maude module specifies a real-time rewrite theory (Σ, E, IR, TR), where:

- (Σ, E) is an equational specification with a signature Σ and E a set of equations. The theory (Σ, E) must contain a specification of a sort `Time` modeling the time domain.
- IR is a set of labeled *instantaneous* (or 'regular') rewrite rules.
- TR is a set of tick rules that model time elapse in the system. Tick rules have the form

  `crl [`*tick*`] : {`S`} ⟶ {`S'`} in time` τ `if` *cond*

  where τ is a term denoting the *duration* of the transition, S and S' are terms of the sort `System`, and `{_}` is an operator (of sort `GlobalSystem`) encapsulating the global state. This form ensures that time advances uniformly in all parts of the system.

The sort `Time` represents a time domain that can be specified by the user [20]. However, for convenience Real-Time Maude provides some useful built-in time domains that can be imported into a specification. In this thesis we use two of them: `POSRAT-TIME-DOMAIN`, the positive rational numbers and `NAT-TIME-DOMAIN-WITH-INF`, the natural numbers plus a special infinity value defined by the constant `INF`. This value is sometimes used when timers are set to be inactive.

Terms of the sort `GlobalSystem` are used to represent the states in Real-Time Maude and should always have the form `{`S`}` where S represents the whole state of the system.

Real-Time Maude extends Maude's object model to support the specification of object-oriented real-time systems. Object-oriented tick rules should generally have the form

```
crl [tick] : {S} => {delta(S, R)} in time R if R <= mte(S) [nonexec] .
```

where the function `mte` determines the most amount of time that can elapse in the system before some critical event must take place, e.g., application of instantaneous rules. The function `delta` defines the effect of time elapse on a system. These functions typically distribute over the elements in the configuration S representing the state of the system.

2.7 Real-Time Maude Analysis

Real-Time Maude not only provides analysis commands similar to those provided by Maude, but it also provides support for the analysis of real-time rewrite theories. Typically, this entails extending existing Maude commands with support for additional time limits. In addition, some new commands such as `find earliest` and `find latest` that can be used to find a state matching a pattern in the shortest or longest duration are provided.

2.7.1 Time Sampling Strategies

To cover the entire time domain (which can be either discrete or dense), tick rules typically have the form `{`t`} => {`t'`} in time` X `if` X`<=`$u \wedge cond$, for X a variable not occurring in t. To execute such rules, Real-Time Maude offers a choice of heuristic-based *time sampling strategies*, so that only *some* moments in time are visited. The choice of such strategies includes:

- Advancing time by a fixed amount Δ in each application of a tick rule.
- The *maximal* strategy, that advances time to the next moment when some action must be taken. That is, time is advanced by u time units in the above tick rule. This corresponds to *event-driven simulation*.

2.7.2 Timed Rewriting

For timed rewriting we use the `trew` and `tfrew` commands. These commands provide timed rewriting capabilities and use the syntax:

`(tfrew` [`[`n`]`] *init* `in time` $\trianglelefteq$ T `.)`
`(tfrew` [`[`n`]`] *init* `with no time limit.)`

where $\trianglelefteq$ is either `<` or `<=` and T is the time bound. The rewrite command applies rewrite rules until the time bound or the bound n on the number of rewrite steps is reached, or no rewrite rule can be applied.

2.7.3 Timed Search

Real-Time Maude's timed search command `tsearch` extends the Maude search command with support for timed systems. This command analyzes all possible behaviors relative to the selected time sampling strategy to find states that match a pattern and condition and are reachable from an initial state within a given duration. The `tsearch` command uses the following syntax:

`(tsearch` [`[`n`]`] *init* `=>*` *pattern* [`such that` *cond*] `in time` $\trianglelefteq$ T `.)`

2.7.4 Searching for Earliest and Latest Solutions

Real-Time Maude provides two new time-specific search commands: `find earliest` and `find latest`. The command `find earliest` obtains the shortest duration that a state can be reached in while matching a pattern that satisfies a condition. The command uses syntax:

`(find earliest` *init* `=>*` *pattern* [`such that` *cond*] `.)`

The command `find latest`, on the other hand, obtains the longest duration possible to reach a state that matches a pattern and uses this syntax

`(find latest` *init* `=>*` *pattern* [`such that` *cond*] `in time` $\trianglelefteq$ T `.)`
`(find latest` *init* `=>*` *pattern* [`such that` *cond*] `with no time limit.)`

2.7.5 Timed Model Checking

Real-Time Maude extends Maude's linear temporal logic model checker so that a time bound can be given to make the state space of a specification finite. There are two versions of the model checker: timed and untimed. The timed model checker supports a time limit and uses syntax:

(mc *initState* |=t *formula* in time $\trianglelefteq$ T .)

Chapter 3

Priced and Priced-Timed Rewrite Theories

A *priced rewrite theory* extends an ordinary *rewrite theory* by assigning a *cost expression* to each rewrite rule. Since priced systems are often also *timed* systems, the definition of priced rewrite theories is further naturally extended to *priced-timed rewrite theories*, which add cost to *real-time rewrite theories* [20]. Priced-timed rewrite theories form the theoretical foundation of Priced-Timed Maude.

Section 3.1, defines the cost domain abstractly. Section 3.2 defines priced rewrite theories. Section 3.3 defines priced-timed rewrite theories as a combination of real-time and priced rewrite theories, thereby allowing us to assign both a cost and duration expression to each rule.

3.1 Cost Domain

Cost is defined abstractly as a commutative monoid:

Definition 3.1.1 *The cost domain is defined abstractly as a commutative monoid* $(\texttt{Cost}, \texttt{0}, \texttt{+}, \texttt{<})$ *with strict total order* `<` *with least element* `0`.

In Maude, this abstract domain can be represented by the following *equational theory*:

```
fth COST is
  sort Cost .
  op 0 : -> Cost .
  op _+_ : Cost Cost -> Cost [assoc comm id: 0] .
  ops _<_ _<=_ : Cost Cost -> Bool .
  vars x, y, z : Cost .
  ceq y = z if x + y == x + z .
  eq x < x = false .
  ceq x < z = true if x < y and y < z .
  eq (x <= y) = (x < y) or (x == y) .
  eq x < 0 = false .
endfth
```

Example 3.1.2 *It is easy to see that the natural numbers* $(\mathbb{N}, 0, + <)$ *satisfy the axioms of above theory* `COST`.

3.2 Priced Rewrite Theories

A *priced rewrite theory* is a a rewrite theory where each rewrite rule is assigned an associated cost expression. Rules where this associated cost always equals `0` are ordinary rewrite rules while the ones with nonzero cost are called *priced rules.*

Definition 3.2.1 *A priced rewrite theory $\mathcal{R}_{\varphi,\kappa}$ is a tuple $(\mathcal{R}, \varphi, \kappa)$ with $\mathcal{R} = (\Sigma, E, L, R)$ a generalized rewrite theory, such that*

- *φ is an equational theory morphism*[1]*.*

  φ : `COST` $\rightarrow$ $(\Sigma,\ E)$.

  That is, the specification (Σ, E) satisfies the requirements of the abstract theory `COST`*, so that φ(`Cost`) is the concrete sort for the cost in (Σ, E) and where the function φ(`_+_`) satisfies the equations for* `+` *in* `COST`*, and so on.*

- *κ is the function that assigns a term $c(x_1, ..., x_n)$ of sort φ(`Cost`) to each labeled rule $l : t(x_1, ..., x_n) \longrightarrow t(x_1, ..., x_n)$.*

Notation: We often write just `Cost`, `0`, and `+` instead of φ(`Cost`), φ(`0`), and φ(`_+_`). Furthermore, we use the following notation for priced rules:

$$l\ :\ t\ \xrightarrow{\kappa(l)}\ t'\ \texttt{if}\ cond$$

where $\kappa(l)$ is the cost incurred by executing the rule l. If $\kappa(l)$ equals φ(`0`) we usually write

$$l\ :\ t\ \longrightarrow\ t'\ \texttt{if}\ cond.$$

The total cost of a (composite) rewrite (proof) $\alpha : t \longrightarrow t'$ is defined as the sum of the cost of each rewrite step in α:

Definition 3.2.2 *Given a composite rewrite $\alpha : t \longrightarrow t'$ in the theory $\mathcal{R}_{\varphi,\kappa}$, the total cost $\kappa^*(\alpha)$ is defined inductively by these rules:*

- Identity*: For each $[t] \in T_{\Sigma,E}$, $\kappa^*([t]) =$* `0`
- Σ-structure*: For each $f \in \Sigma_n$, $n \in \mathbb{N}$, $\kappa^*(f(\alpha_1, \ldots, \alpha_n)) = \kappa^*(\alpha_1)$* `+` *$\ldots$* `+` *$\kappa^*(\alpha_n)$*
- Replacement*: For each rewrite rule $l \in (L, R)$, $\kappa^*(\gamma(\alpha_1, \ldots, \alpha_n)) = \kappa^*(\gamma)$* `+` *$\kappa^*(\alpha_1)$* `+` *$\ldots$* `+` *$\kappa^*(\alpha_n)$*
- Composition*: $\kappa^*(\alpha; \beta) = \kappa^*(\alpha)$* `+` *$\kappa^*(\beta)$*

In [15], Meseguer defines the notion of *proof equivalence* between rewrite proofs. We show that the definition of cost is well-defined in the sense that equivalent rewrite proofs have the same cost:

Theorem 3.2.3 *Given a priced rewrite theory $\mathcal{R}_{\varphi,\kappa}$, then $\kappa^*(\alpha) = \kappa^*(\beta)$ if the proofs α and β are equivalent proofs according to the definition of proof equivalence given in [15].*

Proof

The proof is by induction on the structure proof of the proof equivalence $\alpha = \beta$:

[1] An equational morphism [20] $\phi : (\Sigma, E) \rightarrow (\Sigma', E')$ maps sorts and operators in (Σ, E) to sorts and terms in (Σ', E') in a consistent way. See [20] for a formal definition of equational theory morphism.

1. Category:
(a) Associativity. For all α, β, and δ, must prove $\kappa^*((\alpha;\beta);\delta) = \kappa^*(\alpha;(\beta;\delta))$

Left-hand side:		Right-hand side:	
$\kappa^*((\alpha;\beta);\delta)$		$\kappa^*(\alpha;(\beta;\delta))$	
$\Updownarrow$	[Composition]	$\Updownarrow$	[Composition]
$\kappa^*(\alpha;\beta) + \kappa^*(\delta)$		$\kappa^*(\alpha) + \kappa^*(\beta;\delta)$	
$\Updownarrow$	[Composition]	$\Updownarrow$	[Composition]
$(\kappa^*(\alpha) + \kappa^*(\beta)) + \kappa^*(\delta)$		$\kappa^*(\alpha) + (\kappa^*(\beta) + \kappa^*(\delta))$	
$\Updownarrow$	[Associativity and commutativity of +]	$\Updownarrow$	[Associativity and commutativity of +]
$\kappa^*(\alpha) + \kappa^*(\beta) + \kappa^*(\delta)$		$\kappa^*(\alpha) + \kappa^*(\beta) + \kappa^*(\delta)$	

The left-hand and right-hand sides are equal.

(b) Identities: Must prove $\kappa^*(\alpha;t) = \kappa^*(t';\alpha)$

Left-hand side:		Right-hand side:	
$\kappa^*(\alpha;t)$		$\kappa^*(t';\alpha)$	
$\Updownarrow$	[Composition]	$\Updownarrow$	[Composition]
$\kappa^*(\alpha) + \kappa^*(t)$		$\kappa^*(t') + \kappa^*(\alpha)$	
$\Updownarrow$	[Identity]	$\Updownarrow$	[Identity]
$\kappa^*(\alpha) + 0$		$0 + \kappa^*(\alpha)$	
$\Updownarrow$	[0 is the identity wrt + in the monoid defining cost]	$\Updownarrow$	[0 is the identity wrt + in the monoid defining cost]
$\kappa^*(\alpha)$		$\kappa^*(\alpha)$	

2. Functoriality:
(a) Preservation of composition.
for all $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$, $\kappa^*(f(\alpha_1;\beta_1,\ldots,\alpha_n;\beta_n)) = \kappa^*(f(\alpha_1,\ldots,\alpha_n);f(\beta_1,\ldots,\beta_n))$

Left-hand side:	
$\kappa^*(f(\alpha_1;\beta_1,\ldots,\alpha_n;\beta_n))$	
$\Updownarrow$	[Σ-structure]
$\kappa^*(\alpha_1;\beta_1) + \ldots + \kappa^*(\alpha_n;\beta_n)$	
$\Updownarrow$	[Composition]
$\kappa^*(\alpha_1) + \kappa^*(\beta_1) + \ldots + \kappa^*(\alpha_n) + \kappa^*(\beta_n)$	

Right-hand side:	
$\kappa^*(f(\alpha_1,\ldots,\alpha_n);f(\beta_1,\ldots,\beta_n))$	
$\Updownarrow$	[Composition]
$\kappa^*(f(\alpha_1,\ldots,\alpha_n)) + \kappa^*(f(\beta_1,\ldots,\beta_n))$	
$\Updownarrow$	[Σ-structure]
$\kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n) + \kappa^*(\beta_1) + \ldots + \kappa^*(\beta_n)$	
$\Updownarrow$	[Using associativity and commutativity of +]
$\kappa^*(\alpha_1) + \kappa^*(\beta_1) + \ldots + \kappa^*(\alpha_n) + \kappa^*(\beta_n)$	

We see that the left-hand and right-hand sides are the same.

(b) Preservation of identities:
Must prove $\kappa^*(f([t_1],\ldots,[t_n])) = \kappa^*([f(t_1,\ldots,t_n)])$

Left-hand side:		Right-hand side:	
$\kappa^*(f([t_1],\ldots,[t_n])) = \kappa^*(t_1) + \ldots + \kappa^*(t_n)$		$\kappa^*([f(t_1,\ldots,t_n)])$	
$\Updownarrow$	[Identity]	$\Updownarrow$	[Identity]
$0 + \ldots + 0$		0	
$\Updownarrow$	[0 is the identity of +]		
0			

We see that the left and right-hand sides are equal.

3. Axioms in E: Must prove that for $t(x_1,...,x_n) = t'(x_1,...,x_n)$ an axiom in E, for all $\alpha_1,...,\alpha_n$, $\kappa^*(t(\alpha_1,...,\alpha_n)) = \kappa^*(t'(\alpha_1,...,\alpha_n))$

Left-hand side:		Left-hand side:	
$\kappa^*(t(\alpha_1,...,\alpha_n))$		$\kappa^*(t'(\alpha_1,...,\alpha_n))$	
$\Updownarrow$	[Σ-structure]	$\Updownarrow$	[Σ-structure]
$\kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n)$		$\kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n)$	

Again the left and right hand sides are equal and the assumption holds.

4. Exchange:
For each rewrite rule $l \in R$, $\kappa^*(\gamma(\alpha_1,\ldots,\alpha_n)) = \kappa^*(\gamma(t_1,\ldots,t_n);t'(\alpha_1,\ldots,\alpha_n))$

Left-hand side:	
$\kappa^*(\gamma(\alpha_1,\ldots,\alpha_n))$	
$\Updownarrow$	[Replacement]
$\kappa^*(\gamma) + \kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n)$	

Right-hand side:	
$\kappa^*(\gamma(t_1,\ldots,t_n);t'(\alpha_1,\ldots,\alpha_n))$	
$\Updownarrow$	[Replacement]
$\kappa^*(\gamma) + \kappa^*(t_1) + \ldots + \kappa^*(t_n) + \kappa^*(t') + \kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n)$	
$\Updownarrow$	[Identity]
$\kappa^*(\gamma) + 0 + \ldots + 0 + 0 + \kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n)$	
$\Updownarrow$	[0 is the identity of +]
$\kappa^*(\gamma) + \kappa^*(\alpha_1) + \ldots + \kappa^*(\alpha_n)$	

Again, the left-hand and right-hand side of the proof are the same. □

3.3 Priced-Timed Rewrite Theories

A *priced-timed rewrite theory* combines the notions of *real-time rewrite theory* and *priced rewrite theory*. This can be done in a straightforward manner as the notions of cost and time are disjoint concepts. Priced-timed rewrite theories enable the specification of priced tick rules — rules with both a duration and a cost. The following definition shows how a priced-timed rewrite theory is the combination of real-time and priced rewrite theory:

Definition 3.3.1 *A* priced-timed rewrite theory $\mathcal{R}_{\varphi,\kappa,\phi,\tau}$ *is a tuple* $(\mathcal{R},\varphi,\kappa,\phi,\tau)$ *such that* $(\mathcal{R},\varphi,\kappa)$ *is a priced rewrite theory and* $(\mathcal{R},\phi,\tau)$ *is a real-time rewrite theory.*

For priced tick rules we use the following notation:

l : {t} $\xrightarrow{\kappa(l),\tau(l)}$ {t'} if *cond*

where $\kappa(l)$ is the cost expression of the rule and $\tau(l)$ is the associated duration expression.

Chapter 4

Specifying Priced-Timed Rewrite Theories in Priced-Timed Maude

Priced-Timed Maude extends Real-Time Maude [1, 2] to support the formal specification and analysis of a useful subclass of priced and priced-timed rewrite theories. This chapter gives an overview of Priced-Timed Maude concepts and specification techniques for flat object-oriented and flat "normal" systems.

This chapter also explains how cost domains are defined, how the state of a system should be represented, how Priced-Timed Maude specifications differ from Real-Time Maude specifications, and how to specify priced rules. The specification techniques given in this chapter only work with flat systems, this is defined more precisely in Section 4.3.2.

The tool itself and some examples can be obtained at the following url:
`http://home.ifi.uio.no/lobendik/PTM/`

4.1 Cost Domains

The cost domain is user-specifiable as a data type that satisfies the abstract theory `COST` in Definition 3.1.1. For convenience, two built-in domains are supplied by Priced-Timed Maude: by importing the module `NAT-COST-DOMAIN`, the sort `Cost` is the natural numbers, while importing `POSRAT-COST-DOMAIN` defines the cost domain to be the positive rational numbers.

Before looking at the specifics of a built-in cost domain, we introduce the modules that define the abstract skeleton of the cost domain. The following module `COST` defines the basic sort `Cost` that is used to represent cost along with some operators on it:

```
fmod COST is
```

First we define sorts for cost and non-zero costs. The sort `CostInf` adds an infinity value `infcost`:

```
  sorts Cost NzCost CostInf .
  subsort NzCost < Cost < CostInf .

  op infcost : -> CostInf [ctor] .
```

The rest of the module defines the basic comparison operators `_cheaper than_` and `_cheaper than or eq_`, and the `_pluss_` operator. The constant `free` represents a zero valued `Cost`, i.e., the identity element of the cost domain.

```
 op _pluss_ : Cost Cost -> Cost [assoc comm prec 33 gather (E e) id: free] .

 op free : -> Cost .
 op _cheaper than_ : Cost Cost -> Bool [prec 37] .
 op _cheaper than or eq_ : Cost Cost -> Bool [prec 37] .

 vars C C' : Cost .

 eq C cheaper than or eq C' = (C cheaper than C') or (C == C') .

 eq C cheaper than infcost = true .
 eq infcost cheaper than C = false .
endfm
```

The next module defines the operation of dividing any `Cost` by two; this is needed when performing a binary search for the cheapest possible state. We return to this when discussing the `binary find cheapest` command.

```
fmod DIVIDE-COST is
 including COST .
 op div2 : Cost -> Cost .

 eq div2(free) = free .
endfm
```

Borrowing heavily from the definition of `Time` in Real-Time Maude [2], the following module defines linear cost:

```
fmod LCOST is
 including COST .

 ops minCost maxCost : Cost Cost -> Cost [assoc comm] .

 vars C C' : Cost .
 ceq maxCost(C, C') = C if C' cheaper than or eq C .
 ceq minCost(C, C') = C' if C' cheaper than or eq C .
endfm
```

Finally, all the modules making up the abstract aspects of `Cost` into one single module `ABSTRACT-COST`. This is simply done for convenience and for general overview, so that only one module needs to be included when a new cost domain is defined.

```
fmod ABSTRACT-COST is
 including LCOST .
 including DIVIDE-COST .
endfm
```

Priced-Timed Maude implicitly defines the equational theory morphism φ described in Section 3.2 by mapping the "abstract" sort `Cost` in the *theory* `COST` to the sort `Cost` in the *module* `COST`, `0` to `free`, `_+_` to `_pluss_`, and `<` to `_cheaper than_`. The user can then define his desired cost domain by defining the domain of the sort `Cost` and the functions `free`, `_pluss_` and `_cheaper than_`. This is done for the natural numbers in the built-in module `NAT-COST-DOMAIN` shown below:

```
fmod NAT-COST-DOMAIN is
 including ABSTRACT-COST .
 protecting NAT .
```

After importing the natural numbers, we define the members of the domain and how the abstract operators are defined on this specific domain. That is, we define all natural numbers to be members of the cost domain, while only non-zero natural numbers are considered non-zero cost:

```
 subsort Nat < Cost .
 subsort NzNat < NzCost .

 vars N N' : Nat .
```

We define the identity element (named `free`) to be the number `0`.

```
 eq free = 0 .
```

The following defines that adding two cost elements is the same as adding two natural numbers:

```
 eq N pluss N' = N + N' .
```

Finally, we define the comparison operator `_cheaper than_` for this domain and how division by 2 is to be handled. The division-by-2 operator is used only in conjunction with the `binary find cheapest` command that is covered in the next chapter.

```
 eq N cheaper than N' = N < N' .
 eq div2(N) = N quo 2 .
endfm
```

After importing the above module, we may use the regular operators `<`, `_+_`, and `0` on the members of the domain instead of `_cheaper than_`, `_pluss_`, and `free`.

4.2 Representing the System State

A new sort `SystemState` is introduced in Priced-Timed Maude. The state of the system should always be of this sort; i.e., this sort should represent the whole system. The sort `SystemState` is a subsort of the Real-Time Maude sort `System`. The module `PRICED-SYSTEM` defines these sorts and subsort relationships:

```
fmod PRICED-SYSTEM is
 including UNTIMED-PRELUDE .
 protecting COST .

 sorts SystemState PricedSystem .
 subsort SystemState PricedSystem < System .

...
endfm
```

Furthermore, all initial terms must be of the form `{`S`}` where S is the initial system state, making it a Real-Time Maude term of sort `GlobalSystem`.

4.3 Priced-Timed Modules

A Priced-Timed Maude specification defines an executable priced-timed rewrite theory. Therefore, a Priced-Timed Maude module is essentially a Real-Time Maude specification which includes a definition of a cost domain and where some rules are *priced* rules. Priced rules are (instantaneous or tick) rules equipped with cost expressions. Two new module types are provided in Priced-Timed Maude to define priced rewrite theories: *ordinary* priced-timed modules and *object-oriented* priced-timed modules. The syntax of an ordinary priced-time module is:

```
ptmod NAME is
  BODY
endptm
```

where *BODY* contains a list of imported modules, sort, and subsort declarations, similar to regular Maude modules. In fact, when a `ptmod` is used in a specification, the module `PRICED-TIMED-SYSTEM` shown below is automatically imported into the specification. This module contains "skeleton" definitions of both `Time` (imported from `TIMED-PRELUDE`) and `Cost` (from `PRICED-SYSTEM`) together with important sorts such as the Real-Time Maude sort `GlobalSystem` and the Priced-Timed Maude sort `PricedTimedSystem`. The specifics of what is imported is discussed further in Chapter 6.

The module `PRICED-TIMED-SYSTEM` makes a specification priced-timed by importing the module `PRICED-SYSTEM` along with the skeleton defining `Time` and timed systems from Real-Time Maude's `TIMED-PRELUDE`:

```
fmod PRICED-TIMED-SYSTEM is
 including PRICED-SYSTEM .
 including TIMED-PRELUDE .

 sort PricedTimedSystem .
 subsort ClockedSystem < PricedTimedSystem .
 op _with cost_ : ClockedSystem Cost -> PricedTimedSystem [prec 95 gather (E e)] .
endfm
```

An *object-oriented* priced-timed module is essentially an object-oriented Real-Timed Maude module with the addition of a cost domain and priced rewrite rules. Object-oriented priced-timed modules are written with the following syntax:

```
ptomod NAME is
  BODY
endptom
```

When a `ptomod` is used in a specification, the module `PRICED-TIMED-OO-SYSTEM` is automatically imported, which in turn imports the Real-Time Maude module `TIMED-OO-PRELUDE` that defines the skeleton of an object-oriented Real-Time Maude module. In addition, the module `PRICED-OO-SYSTEM` is imported and states the crucial subsort declaration:

```
subsort Configuration < SystemState .
```

This declaration means that the sort `Configuration` is the state of the system and the whole state of the system. Furthermore, the module `PRICED-TIMED-OO-SYSTEM` defines the skeletons of the functions used in object-oriented tick rules; this is covered in Section 4.4.

4.3.1 Priced Rules

A priced rule

$$l \ : \ t \ \xrightarrow{\kappa(l)} \ t' \ \texttt{if} \ cond$$

is written with syntax

$$\texttt{crl [}l\texttt{] : } t \texttt{ => } t' \texttt{ with cost } \kappa(l) \texttt{ if } cond \texttt{ .}$$

in Priced-Timed Maude. The terms t and t' are of sort `SystemState` and the term $\kappa(l)$ of sort `Cost`. A *priced* tick rule

$$l \ : \ \{t\} \ \xrightarrow{\kappa(l),\tau(l)} \ \{t'\} \ \texttt{if} \ cond$$

is written with syntax

$$\texttt{crl [}l\texttt{]: } \{t\} \texttt{ => } \{t'\} \texttt{ in time } \tau(l) \texttt{ with cost } \kappa(l) \texttt{ if } cond \texttt{ .}$$

4.3.2 Flat Systems

As previously stated, the current prototype of Priced-Timed Maude only works with *flat* systems. This limitation is purely due to implementation concerns and not theoretical ones. Future versions of the tool should be able to overcome this constraint.

The reasons for the flat system requirement are:

1. Limited implementation time; supporting non-flat systems are too complex to implement inside our time frame.
2. All useful examples encountered so far have been flat systems.

However, these specification techniques are still useful as flat systems include a large and interesting class of systems: flat object-oriented systems.

The following definition tells us what is considered a flat Priced-Timed Maude specification:

Definition 4.3.1 (Flat Systems) *The priced rules of a Priced-Timed Maude systems must have the form*

$$\texttt{crl [}l\texttt{] : } t \texttt{ => } t' \texttt{ with cost } \kappa(l) \texttt{ if } cond \texttt{ .}$$

for the system to be considered flat all the priced rules must be able to be rewritten as

$$\texttt{crl [}l\texttt{] : } \{t\} \texttt{ => } \{t'\} \texttt{ with cost } \kappa(l) \texttt{ if } cond \texttt{ .}$$

or for object oriented systems

$$\texttt{crl [}l\texttt{] : } \{C\ t\} \texttt{ => } \{C\ t'\} \texttt{ with cost } \kappa(l) \texttt{ if } cond \texttt{ .,}$$

where t is a term of the sort `SystemState` and the operator `{_}` is defined as:

```
op {_} : System -> GlobalSystem [format (g o g so)] .
```

and C is a variable of sort `Configuration` *that in conjunction with t represent the whole system.*

That is, the term {t} represents the whole system, or the term {C t} represents the whole system. Priced tick rules are already on this form. This ensures flat application of priced rules and priced tick rules.

Generally, this means that for object-oriented priced-timed specifications where we want to apply priced rules to specific objects, these objects must reside in the outer configuration encapsulated by the `{_}` operator.

4.4 Object-oriented Tick Rules and Built-in Functions

When we work with an object-oriented priced-timed module, Priced-Timed Maude automatically imports *skeletons* of the functions `mte`, `delta`, and `rate`. We recall `mte` and `delta` from Real-Time Maude, while `rate` is a new function. The function `mte` computes the most amount of time that can elapse in an application of the function and `delta` defines how the elapse of time affects the system. The function `rate` defines the cost of waiting one time unit. The skeletons of these functions are automatically imported so that the user does not need to deal with declaration of their operators and basic equations. Consequently, the user only needs to give equations defining behavior of the functions for the objects used in the specific system.

As previously mentioned, the following functional module is imported into a specification when it is object-oriented:

```
mod PRICED-TIMED-OO-SYSTEM is
```

The included modules define a skeleton for the time domain as well as some essential sorts needed by the functions `delta,`, `mte`, and `rate`.

```
including LTIME-INF .
including PRICED-OO-SYSTEM .
including TIMED-OO-PRELUDE .
including PRICED-TIMED-SYSTEM .

var R : Time .
vars NeC NeC' : NEConfiguration .
var C : Cost .
```

The basic skeletons of all three functions are designed to take the whole `Configuration` representing the system as an argument, then split this down to distinct objects. The following defines the basic skeleton of the function `delta`:

```
op delta : PricedSystem Time
           -> PricedSystem [frozen (1)] .
eq delta(NeC with cost C, R) = delta(NeC, R) with cost C .
eq delta(NeC NeC', R) = delta(NeC, R) delta(NeC', R) .
eq delta(none, R) = none .
```

The equations defining `delta` for each class in the specification must then be given by the user and should have the following form:

```
eq delta(< O : Class | Atts >, R) =  < O : Class | f(Atts,R) > .
```

where *Atts* is the object's set of attributes, R is a variable of sort `Time`, and f is some function that alter these in some way. Typically, f increases or decreases clocks and timers in the set of attributes by the value of `R`.

The following defines the basic skeleton for the function:

```
op mte : PricedSystem -> TimeInf [frozen (1)] .
eq mte(NeC with cost C) = mte(NeC) .
eq mte(NeC NeC') = minimum(mte(NeC), mte(NeC')) .
eq mte(none) = INF .
```

The equations defining the function `mte` for the different classes in the system should also be supplied by the user and have the form:

```
eq mte(< O : Class | Atts >) =  g(Atts) .
```

The function g extracts the most allowable time to pass from the object. Typically, this is the time until the next timer expires.

A rate function is commonly used in priced-timed systems to determine the rate of cost incurred per time unit. The following defines the skeleton of the `rate` function:

```
 op rate : PricedSystem -> Cost [frozen (1)] .
 eq rate(NeC with cost C) = rate(NeC) .
 eq rate(NeC NeC') =
    rate(NeC) pluss rate(NeC') .
 eq rate(none) = free .
endm
```

The `rate` function is usually based on the attributes of an object. An equation of the following form is used when a rate for a class is defined:

```
eq rate(< O : Class | Atts >) =  h(Atts) .
```

Tick rules in an object-oriented system typically have the form:

```
crl [tick]: {S} => {delta(S,T)} in time T with cost rate(S) * T if T <= mte(S) .
```

Sometimes we may refer to the above rule as the standard Priced-Timed Maude object-oriented tick rule.

4.5 Simple Examples of Priced-Timed Maude Specifications

This section presents some examples to illustrate how to specify a system in Priced-Timed Maude.

4.5.1 Priced Thermostat

With today's prohibitive energy prices, it is useful to determine the cost of keeping a target temperature in our living rooms. By slightly modifying the thermostat specification from [20], we now have a tool to help achieve this.

When we add the power consumption cost to the rules of the thermostat specification, it works in this manner: we want to model a thermostat that keeps the temperature in a room between 62 and 72 degrees by turning on and off a heater. When the heater is off, the room temperature decreases by one degree every minute. Conversely, at 62 degrees the heater should be turned on, consuming 50 watt-minutes of power. While in this state, the heater consumer power at a rate of 100 watts and the temperature increases by a rate of two degrees per minute.

The following Priced-Timed Maude module models the new thermostat:

```
(ptmod PRICED-THERMOSTAT is
```

We import the appropriate value domains: the positive rational numbers for both time and cost:

```
protecting POSRAT-TIME-DOMAIN .
protecting POSRAT-COST-DOMAIN .
```

The following tells Priced-Timed Maude that the thermostat is the whole system:

```
sort Status Thermostat .
subsort Thermostat < SystemState .
```

The following operators define the thermostat as a pair of a status and a temperature represented as a positive rational number:

```
ops on off : -> Status [ctor] .
op _‘,_ : Status PosRat -> Thermostat [ctor] .
```

It is typical for several kinds of electrical equipment, that do not hold an initial charge (as opposed to a piece of equipment on standby), to consume some power when turned on. The rule `turn-on` switches the thermostat on at 62 degrees, consuming 50 watt-minutes of power:

```
rl [turn-on] :   off , 62  =>  on , 62  with cost 50 .
```

While `turn-off` switches the thermostat off at 74 degrees:

```
rl [turn-off] :  on , 74  =>  off , 74 .
```

The tick rule `tick-on` makes sure power is consumed at a rate of 100 power units per time unit and the temperature is increased by 2 degrees per minute when the heater is `on`, while at the same time making sure time does not advance past a point where the thermostat should be turned off:

```
vars R R' : Time .
var T : PosRat .

crl [tick-on] :
    {(on, T)} => {(on, T + (2 * R'))} in time R' with cost R' * 100
if R' <= ((74 - T) / 2) [nonexec] .
```

The tick rule `tick-off` ensures the temperature falls by the appropriate amount when the heater is `off`, while making sure time does not advance beyond a moment the thermostat should be turned on:

```
    crl [tick-off] :
        {(off, T)} => {(off, T - R')} in time R' if R' <= (T - 62) [nonexec] .
  endptm)
```

4.5.2 Concurrent Priced-timed Lights

In this example we model a system with timed light switches each activated by a sensor. Each light switch is connected to a light bulb with a given wattage. When a switch is turned on, power equal to the wattage of the bulb running for one time unit is consumed and a timer is set to count down five time units. While a light switch is turned on, power is consumed at a rate equal to the wattage of the fitted bulb while the timer is counted down until it reaches zero when it is turned off. A switch that is off consumes no power. A switch may stay off indefinitely.

The following object-oriented Priced-Timed Maude module models the system described above:

```
(ptomod PRICED-TIMED-OO-LIGHT-SWITCH is
  protecting NAT-TIME-DOMAIN-WITH-INF .
  protecting NAT-COST-DOMAIN .
```

The class used to represent the switch itself contains attributes for the status (`on`/`off`) of the lights, the wattage of the bulb used and the timer:

```
    sort Status .
    ops on off : -> Status [ctor] .

    class Switch | status : Status, wattage : Cost, timer : TimeInf  .

    var O : Oid .
    var W : Cost .
    var Ti : TimeInf .
    vars T R : Time .
    var S : SystemState .
```

The rule `turn-off` turns the switch off when the timer reaches 0 by setting its status to `off` and the timer to `INF`:

```
    rl [turn-off] : < O : Switch | status : on, timer : 0 >
                    =>
                    < O : Switch | status : off, timer : INF > .
```

When the rule `turn-on` is executed, cost equal to the wattage of the bulb is incurred and timer of the switch is set to 5 and its status is set to `on`:

```
    rl [turn-on] : < O : Switch | status : off, wattage : W, timer : INF >
                   =>
                   < O : Switch | status : on, timer : 5 > with cost W .
```

The function `mte` returns the state of the `timer` attribute as this is the next moment in time we cannot tick past, but have to turn the `Switch` off:

```
    eq mte(< O : Switch | status : on, timer : Ti >) = Ti .
    eq mte(< O : Switch | status : off >) = INF .
```

The function `delta` decreases the `timer` attribute by the appropriate amount simulating the flow of time if a switch is on. When a switch is off it is unaffected by the flow of time.

```
    eq delta(< O : Switch | status : on, timer : Ti >, R) =
             < O : Switch | timer : Ti monus R > .
    eq delta(< O : Switch | status : off >, R) =
             < O : Switch | > .
```

The function `rate` reads the wattage of the light bulb used if the switch is on. Obviously switches that are off do not consume any power:

```
    eq rate(< O : Switch | status : on, wattage : W >) = W .
    eq rate(< O : Switch | status : off >) = 0 .
```

The rule `tick` is the standard Priced-Timed Maude object-oriented tick rule:

```
    crl [tick] : {S} => {delta(S, R)} in time R with cost (rate(S) * R)
                 if R <= mte(S) [nonexec] .
  endptom)
```

The following module defines 2 initial states `init1` and `init2`. The initial state `init1` has one switch named `Driveway` with one 40-watt bulb. The second initial state `init2` has 2 lights. In addition to the aforementioned driveway light, it also has a switch named `Garden` with a 25-watt bulb:

```
  (ptomod TEST-TWO-LIGHTS is
    protecting PRICED-TIMED-OO-LIGHT-SWITCH .
    protecting STRING .

    subsort String < Oid .

    ops init1 init2 : -> GlobalSystem .
    ops driveway garden : -> Configuration .

    eq driveway = < "Driveway" : Switch | status : off, wattage : 40, timer : INF > .
    eq garden = < "Garden" : Switch | status : off, wattage : 25, timer : INF > .
    eq init1 = {driveway} .
    eq init2 = {driveway garden} .
  endptom)
```

Chapter 5

Analysis in Priced-Timed Maude

This chapter gives an overview of Priced-Timed Maude's analysis capabilities and how they differ from their Real-Time Maude counterparts.

Priced-Timed Maude provides the following analysis methods for priced-timed specifications:

- *Rewriting*: priced-timed rewriting can be used to simulate one behavior of a system from an initial state within a time limit, cost limit, and/or number of rewrite steps.
- *Search*: priced-timed search extends the timed search provided by Real-Time Maude with an additional cost limit to restrict the search space.
- *Search for optimal solutions*: although not primarily designed as an optimization tool, Priced-Timed Maude provides additional search capabilities to obtain the cheapest cost or the shortest duration needed to reach a state.
- *Linear temporal model checking* is used to check temporal logic properties for all possible behaviors from a given initial state. Real-Time Maude's model checker is extended by Priced-Timed Maude to analyze priced-timed specifications, but not to check cost properties.

Priced-Timed Maude extends Real-Time Maude's methods/commands for priced-timed systems; e.g., in their use of the selected time sampling strategy. Although the Real-Timed Maude versions of these commands could in some cases be used, they are not guaranteed to work with Priced-Timed Maude specifications. In most cases the Priced-Timed Maude versions of these commands have to make theory transformations on the specifications and initial terms to translate them into Real-Time Maude specifications and initial terms. The reason for this is that Real-Time Maude does not "understand" the concept of cost and priced rules. In addition, most Priced-Timed Maude counterparts of Real-Time Maude commands incorporate additional limits on cost.

Although Priced-Timed Maude supports specification and analysis of priced systems without time involved, this chapter discusses only the commands that are used to analyze priced-timed systems. This is because we did not encounter any useful examples that could be specified as priced systems. In addition, the priced-timed versions of the commands are mostly extended versions of the non-timed versions. Therefore, only the priced-timed versions are discussed to avoid repeating the description of each command.

5.1 Executing Priced-Timed Maude Specifications

To start Priced-Timed Maude, the file `priced-timed-maude.maude` needs to be executed by the Maude interpreter.

The following sections describe the different analysis commands provided by Priced-Timed Maude. All examples using the `PRICED-THERMOSTAT` specification are assumed to use default time increase 1/5 and those using the `PRICED-TIMED-OO-LIGHT-SWITCH` specification are assumed to use default time increase 1 unless stated otherwise.

5.2 Priced-Timed Rewriting

Priced-Timed Maude's priced rewrite commands extend the timed rewriting capabilities provided by Real-Time Maude to priced-timed systems. This entails adding an optional cost limit to the rewrite sequence. Priced-Timed Maude makes sure that the total cost in the system does not exceed the given limit during the rewrite sequence. For instance, in our thermostat example, we might be interested in knowing if keeping our living room warm will raise our electricity bill beyond a certain level.

Two strategies for priced-timed rewriting are provided: `ptrew` and `ptfrew`. The `ptfrew` command is an extension of the `tfrew` Real-Time Maude command and `ptrew` is an extension of the `trew` command. The syntax of both commands is similar, therefore, only the syntax for `ptfrew` is shown:

`(ptfrew` [[n]] *initialState* `in time` $\trianglelefteq$ T `with cost` $\trianglelefteq$ C `.)`

For rewriting with no time limit, but a cost limit we use:

`(ptfrew` [[n]] *initialState* `with no time limit with cost` $\trianglelefteq$ C `.)`

For rewriting with a time limit, but with no cost limit we use:

`(ptfrew` [[n]] *initialState* `in time` $\trianglelefteq$ T `with no cost limit.)`

Finally, for rewriting with no cost or time limit we use:

`(ptfrew` [[n]] *initialState* `with no limits.)`

The '[n]' part is optional, with n giving an upper limit on the number of rewrite steps, *initialState* is some initial term of sort `GlobalSystem`, T is a term of sort `Time`, C is a term of sort `Cost`, and $\trianglelefteq$ is either `<` or `<=`.

Example 5.2.1 *We simulate the behavior of a driveway light using the module* `PRICED-TIMED-OO-LIGHT-SWITCH` *by using the rewrite commands on the initial state* `init1` *given in the module* `TEST-TWO-LIGHTS` *in Section 4.5.2. This is the initial state containing only the light switch named* `Driveway` *with a 40-watt bulb installed.*

We simulate the behavior of the system for 15 minutes with the following command:

```
Maude> (ptfrew init1 in time <= 15 with no cost limit.)

Result PricedTimedSystem :
  {< "Driveway" : Switch | status : on, timer : 2, wattage : 40 >}
   in time 15 with cost 640
```

We see that the light consumed 640 power units in 15 minutes.

Example 5.2.2 *In this example we simulate the thermostat system in the module* `PRICED-THERMOSTAT`. *If we are on a tight budget, we might want to know how long our thermostat can be on before the power bill becomes too expensive. We now want to know how long the thermostat can be on in a day if we want it to consume no more than 6000 watt-minutes. As the thermostat is a deterministic system, we can simply perform a rewrite up until the given cost limit to determine this:*

```
Maude> (ptfrew {off, 62}  with no time limit with cost <= 6000 .)

Result PricedTimedSystem :
  {on,64} in time 163 with cost 6000
```

That is, keeping the room temperature between 62 and 74 degrees for 2 hours and 43 minutes consumes 6000 watt-minutes.

5.3 Priced-Timed Search

Priced-Timed Maude extends Real-Time Maude's search capabilities with the ability to analyze priced-timed specifications and add cost limits to restrict the state space to be searched. Searching with a cost limit may be useful when a budget must be satisfied. Internally, priced search is implemented by a transforming a priced-timed specification into a Real-Time Maude specification and passing it along to Real-Time Maude's search function which then performs the search.

The most basic extension of the timed search command is the priced-timed search command `ptsearch`. There are 4 versions of this command: search with time and cost limit, only time limit, only cost limit, and neither time nor cost limit. The following shows the syntax for the 4 versions. For a search with time and cost limits we use:

```
(ptsearch [[n]] initialState =>* searchPattern [such that cond]
          in time ⊴ T with cost ⊴ C .)
```

For a search with no time limit but a cost limit we use:

```
(ptsearch [[n]] initialState =>* searchPattern [such that cond]
          with no time limit with cost ⊴ C .)
```

For a search with a time limit but no cost limit we use:

```
(ptsearch [[n]] initialState =>* searchPattern [such that cond]
          in time ⊴ T with no cost limit.)
```

For a search without time and cost limit we use:

```
(ptsearch [[n]] initialState =>* searchPattern [such that cond]
          with no limits .)
```

Again, '[n]' is optional with n the upper bound on the number of solutions to display. The initial state *initialState* is a ground term of sort `GlobalSystem`. *searchPattern* is a normal Maude search pattern that may contain variables. [`such that` *cond*] is optional and the *cond* part may be conditions on the variables in *searchPattern*. T is a term of sort `Time` and C is a term of sort `Cost`.

Once the desired number of matches satisfying both the pattern and the conditions is found (or there are no more matches), the found solutions with the appropriate substitutions are displayed. The following two substitutions will always be displayed: `TIME_ELAPSED` denoting the amount of time lapsed to reach the solution and `TOTAL_COST_INCURRED` denoting the cost of reaching the solution.

Example 5.3.1 *We can use the search command to check if it is possible that the thermostat will ever be off at a temperature below 62 degrees when it starts off at 62 degrees with no cost or time limit:*

```
Maude> (ptsearch [1] {off, 62} =>* {off, P:PosRat} such that P:PostRat < 62
                                  with no limits.)

No more solutions
```

Unsurprisingly, this is not possible.

Example 5.3.2 *If we want to know if the room temperature reaches 68 degrees in 5 minutes we could search for the following:*

```
Maude>(ptsearch [1] {off, 62} =>* {S:Status, 68} in time <= 5 with no cost limit.)

Solution 1
S:Status --> on ; TIME_ELAPSED:Time --> 3 ; TOTAL_COST_INCURRED:Cost --> 350

No more solutions
```

Example 5.3.3 *Similarly, we can check if it is possible that our driveway lights can turn themselves off in 2 minutes if they have just been activated*

```
Maude> (ptsearch [1] {< "Driveway" : Switch | status : on, wattage : 40,
                                              timer : 5 >}
                     =>*
                     {< "Driveway" : Switch | status : off >}
                     in time <= 2 with no cost limit.)

No more solutions
```

This search has no solutions as one might expect.

5.4 Search for Optimal Solutions

In addition to basic priced-timed search, Priced-Timed Maude provides a new command to find the cheapest state matching a search pattern, condition, and time limit. Furthermore, the Real-Time Maude command to find the earliest state matching a pattern and a condition is extended to accept a cost limit and to work properly with priced-timed systems.

5.4.1 Find Cheapest

Priced-Timed Maude provides the user with a way to find the cheapest possible state matching a pattern within a time limit. This can be useful when we need to find the cheapest way to achieve something like, for instance, the cheapest schedule within a certain time limit or the cheapest way to keep our home at a certain temperature.

Two commands are provided for this purpose: `find cheapest` and `binary find cheapest`. They differ only internally in the way they obtain the result. Both start out by trying to find a state that matches the pattern and criteria, then incrementally searches for better solutions until the cheapest one is obtained. The binary version uses a binary search algorithm and the "normal" version uses a lazier algorithm (as

explained in Chapter 6). The `find cheapest` commands are useful for smaller problems, but for larger problems execution time quickly becomes a factor. This is because multiple searches are performed sequentially to obtain the optimal solution in the implementation of the commands.

Termination of both versions of this command is not guaranteed unless the cost domain is *well-founded*. Moreover, for `binary find cheapest` to work, the module `DIVIDE-COST` that defines division of cost by 2 needs to be imported as part of the specification's cost domain. Like the `ptsearch` command, the `find cheapest` commands employ the built-in Priced-Timed Maude search capabilities.

The two commands use similar syntax as follows (add `binary` in front of `find` for the binary version):

`(find cheapest` *initialState* `=>*` *searchPattern* [`such that` *cond*] `in time` $\trianglelefteq$ T `.)`

`(find cheapest` *initialState* `=>*` *searchPattern* [`such that` *cond*] `with no time limit .)`

Example 5.4.1 *We can use the* `find cheapest` *command to obtain the least possible power consumption needed to reach 70 degrees from a state where the thermostat is at 62 degrees and off:*

```
Maude> (find cheapest {off, 62} =>* {S:Status, 70} with no time limit .)

Solution
S:Status --> on ; TIME_ELAPSED:Time --> 4 ; TOTAL_COST_INCURRED:Cost --> 450
```

This shows that the cheapest cost at which the desired state is reachable is in 4 minutes and has cost 450 power units.

Example 5.4.2 *Likewise we find the cheapest way our driveway lights can turn themselves off from just being activated:*

```
Maude> (find cheapest {< "Driveway" : Switch | status : on, wattage : 40,
                                               timer : 5 >}
                      =>*
                      {< "Driveway" : Switch | status : off >}
                      with no time limit .)

Solution
CLASS_OF_"Driveway":Switch --> Switch ;
REMAINING_ATTRIBUTES_OF_"Driveway":AttributeSet --> timer : INF, wattage : 40  ;
TIME_ELAPSED:Time --> 5 ; TOTAL_COST_INCURRED:Cost --> 200
```

Unsurprisingly, the cheapest state in which the lights have been turned off is after 5 minutes, consuming 200 power units.

5.4.2 Priced Find Earliest

In some cases it might be useful to find a state that matches some criteria and can be reached in the shortest amount of time. For instance, what is the least amount of time it takes to heat the living room from 62 to 70 degrees? The Real-Time Maude command `find earliest` accomplishes this. The command has been adapted for Priced-Timed Maude to accept an optional price limit and can be used with priced-timed specifications via the command `priced find earliest`.

This command starts by finding a state that matches the criteria, then tries to find earlier solutions, when there are no earlier solutions to be found the last to be found is displayed. The two versions of `priced find earliest` has the following syntax:

```
(priced find earliest initialState =>* searchPattern [such that cond]
                                    with no cost limit .)

(priced find earliest initialState =>* searchPattern [such that cond]
                                    with cost ⋈ C  .)
```

Example 5.4.3 *Using the* `priced find earliest` *command we can easily verify that the earliest our lights can be turned off if they have just been turned on is after 5 minutes*

```
Maude> (priced find earliest  {< "Driveway" : Switch | status : on, wattage : 40,
                                                       timer : 5 >}
                              =>*
                              {< "Driveway" : Switch | status : off >}
                              with no cost limit.)

Result: {< "Driveway" : Switch | status : off, wattage : 40, timer : INF >}
        in time 5 with cost 200
```

Example 5.4.4 *We can also verify that the cheapest way of reaching 70 degrees from 62 and off is also the earliest:*

```
Maude> (priced find earliest  {off, 62} =>* {S:Status, 70}
                                           with no cost limit.)

Result: {on,70} in time 4 with cost 450
```

5.5 Temporal Logic Model Checking

Priced-Timed Maude provides access to Real-Time Maude's temporal logic model checker to check temporal logic properties for a Priced-Timed Maude specification.

There are no differences from the Real-Time Maude model checker apart from the ability to model check Priced-Timed Maude systems. This means that an atomic proposition p for the whole system state holds whether there is a cost present in the system or not. That is, cost is ignored when checking if an atomic proposition holds. To use the model checker, the command `pmc` is used with similar syntax as the Real-Time Maude model checker invoked by the command `mc`:

```
(pmc initialState |=t formula with no time limit .)
```

or

```
(pmc initialState |=t formula in time ⊴ T .)
```

The *formula* is a temporal logical formula built from atomic propositions, the normal logical operators, and the temporal logical operators `[]`, `<>`, `U`, etc.

The untimed model checker is invoked with the following syntax:

```
(pmc initialState |=u formula .)
```

When declaring atomic propositions for the priced model checker, the module `PRICED-MODEL-CHECKER` needs to be imported. This makes sure that propositions hold whether or not they are associated with a cost.

Example 5.5.1 *In this example we declare some atomic propositions for the priced thermostat. The following module imports the priced thermostat and the priced model checker and declares 3 atomic propositions:*

```
(ptmod MODEL-CHECK-THERMOSTAT is
  protecting PRICED-MODEL-CHECKER .
  protecting PRICED-THERMOSTAT .

  ops therm-on therm-off : -> Prop [ctor] .
  op temp-is : PosRat -> Prop [ctor] .

  vars T T' : PosRat .
  var S : Status .
```

The proposition `therm-off` *is true when the heater is off,* `therm-on` *is true when the heater is on, and the final proposition* `temp-is(T')` *when the temperature T of the thermometer is the same as T'.*

```
  eq {off, T} |= therm-off = true .
  eq {on, T} |= therm-on = true .
  eq {S, T} |= temp-is(T') = (T == T') .
endptm)
```

Example 5.5.2 *We test the assumption that the heater will eventually be turned on after the thermostat has reached 62 degrees with the heater off:*

```
Maude> (pmc {off, 62} |=u <> therm-on .)

Result Bool:
true
```

Example 5.5.3 *We test the assumption that when the heater is turned on at 62 degrees, it will not be turned off until the room temperature reaches 72 degrees:*

```
Maude> (pmc {on, 62} |=u therm-on U therm-is(72) .)

Result Bool:
true
```

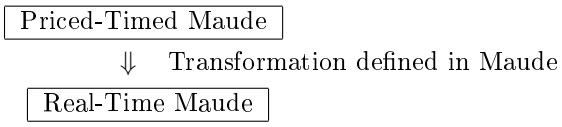
Model checking of cost properties was not found to be particularly useful and was therefore not implemented. The model checker is not used in the rest of this thesis, therefore, this topic will not be discussed further.

Chapter 6

Semantics of Priced-Timed Maude

This chapter presents the semantics of Priced-Timed Maude by showing how a Priced-Timed Maude module is represented in Real-Time Maude and by showing how Priced-Timed Maude commands are translated into Real-Time Maude commands. One of the main goals of Priced-Timed Maude is to support the specification and analysis of priced and priced-timed systems. Initially, support for both kinds of systems was planned. However, because I have not seen any interesting untimed priced systems, the priced-timed aspects have been favored. Some support for regular priced systems still remains; however, priced-timed systems is the focus of this chapter.

Priced-Timed Maude has been implemented in Maude as an extension of Real-Time Maude using Maude's powerful reflective and meta-programming capabilities. One design goal for Priced-Timed Maude is to extend Real-Time Maude rather than to alter it. This goal is met by defining a Real-Time Maude semantics of Priced-Timed Maude commands, so that the Priced-Timed Maude commands are executed by transforming the Priced-Timed Maude module and command into a pair of a Real-Time Maude module and command:



In fact, all of Priced-Timed Maude is implemented by defining transformations in Maude that translate Priced-Timed Maude specifications, terms, and commands into Real-Time Maude representations of these.

6.1 Overview of the Priced-Timed Maude Semantics

The implementation of Priced-Timed Maude relies on the fact that Priced-Timed Maude modules and terms can be meta-represented as terms in Maude. Therefore, these can be manipulated and transformed by Maude functions.

In Priced-Timed Maude, we work with modules on *three* distinct levels: *user input*, *database*, and *execution*. *User input* corresponding to a Priced-Timed Maude module M is first parsed according to the grammar in Section 6.4, and is stored in Real-Time Maude's module *database* as a Real-Time Maude module $M_{\texttt{RTM}}$. When a command is executed, $M_{\texttt{RTM}}$ is transformed to a *different* Real-Time Maude module $M'_{\texttt{RTM}}$, depending on the command to be executed.

Section 6.1.1 explains how Priced-Timed Maude modules are translated from user input to Real-Time Maude modules in Real-Time Maude's module database. To make parsing of user input as simple as possible, this first translation is essentially the "identity translation" from Priced-Timed Maude into

Real-Time Maude, and is achieved by just turning the *language construct* `with cost` in a Priced-Timed Maude rule into a *function symbol* `with cost` in the Real-Time Maude representation.

Section 6.1.2 discusses the main idea behind command execution in Priced-Timed Maude, which involves a series of transformations. Specifically, Priced-Timed Maude commands and modules (stored as Real-Time Maude modules in the database) are transformed into Real-Time Maude commands and modules. As mentioned, this involves a somewhat different module transformation than in the first step. Section 6.1.3 explains how during command execution, a "state" in Priced-Timed Maude is represented in Real-Time Maude as a term {S `with cost` C}, where S is the state and C is the cost accumulated to reach that state. Section 6.1.4 discusses how priced rules are handled during command execution. This is done by translating them from the form t `=>` t' `with cost` c, into a Real-Time Maude rule having the form {t `with cost OLDCOST`} `=>` {t' `with cost OLDCOST pluss` c}, where `OLDCOST` is a new variable and the cost of the left-hand side state. In object-oriented specifications, the above priced rule is transformed into a rule

```
{t C:Configuration with cost OLDCOST}
=>
{t C:Configuration  with cost OLDCOST pluss c} .
```

6.1.1 Parsing an Storing Priced-Timed Maude Modules

A *different* representation than the *user input* is used for storing a module in the Real-Time Maude *database*. This representation, is in fact "the identity" transformation from Priced-Timed Maude to Real-Time Maude, and is achieved by importing certain operators and sorts into the module while parsing it according to the grammar in Section 6.4,. Specifically, priced rules

```
crl S => S' with cost C .
```

are transformed by importing the following:

```
sorts SystemState PricedSystem .
subsorts SystemState PricedSystem < System .

op _with cost_ : SystemState Cost -> PricedSystem [prec 95 gather (E e)] .
```

The rule is otherwise "unchanged." This transforms the *syntactic construct* `with cost` in the priced rules into the *function symbol* `_with cost_` in the Real-Time Maude database representation of the priced rule. Priced *tick* rules

```
crl {S} => {S'} in time T with cost C if cond .
```

are transformed in a similar manner by importing the following into the module:

```
sort PricedTimedSystem .
subsort ClockedSystem < PricedTimedSystem .

op _with cost_ : ClockedSystem Cost -> PricedTimedSystem [prec 95 gather (E e)] .
```

Again, the *syntactic construct* `with cost` in the priced rules is transformed into the *function symbol* `_with cost_` in the Real-Time Maude database representation of the priced rule.

However, when commands are executed priced and priced tick rules are transformed further. The reason for this "intermediate" representation of a Priced-Timed Maude module in Real-Time Maude database is that this makes the task of parsing user input as simple as possible as the Real-Time Maude implementation of user input parsing is simply reused.

6.1.2 Semantics of Priced-Timed Maude Commands

Command execution in Priced-Timed Maude uses the same idea as Real-Time Maude's command execution. The main idea is to transform a Priced-Timed Maude module[1] and command to Real-Time Maude module *and* command:

$$\begin{array}{c}(command_{\texttt{PTM}}(t_0, Args), module_{\texttt{RTM}}) \\ \Downarrow \alpha \\ (command_{\texttt{RTM}}(\gamma(t_0), Args), \lambda(module_{\texttt{RTM}}))\end{array}$$

The transformation α is a Maude function `procPriceTimedCommand` that performs a series of transformations[2]. This series transforms the arguments and Priced-Timed Maude command into proper Real-Time Maude arguments and a Real-Time Maude command. All the transformations are discussed in Section 6.2.

6.1.3 Representation of States

As mentioned, when a command is executed, the representation of the module is changed. In addition, the representation of the state is changed to a Real-Time Maude state. The state of a Real-Time Maude system is represented as a `GlobalSystem` term `{`S`}` where S is a term of the sort `System`. When executing a command in Priced-Timed Maude the state of a system is also represented as a `GlobalSystem` term, but of the form `{`S `with cost` C`}`, where the operator `_with cost_` is *now* defined to take the system state S as its first argument and the global cost C as its second:

```
op _with cost_ : SystemState Cost -> PricedSystem [prec 95 gather (E e)] .
```

The following subsort declaration makes this possible:

```
subsorts SystemState PricedSystem < System .
```

As mentioned in Section 6.1.1, this `_with cost_` operator and the appropriate sort, and subsort declarations are automatically imported into the module during the parsing process.

6.1.4 Priced Rules

At the command execution stage, Priced-Timed Maude deals with modules stored in the Real-Time Maude database. When encountering a priced rule t `=>` t' `with cost` c it is (further) transformed into a rule having the form
`{`t `with cost OLDCOST}` `=>` `{`t' `with cost OLDCOST pluss` c`}`. This means that after the translation each application of a priced rule simply 'deposits' the cost of performing the rule into the state of the system.

A priced instantaneous rule

`crl [`l`] : {`S`} => {`S'`} with cost` C `if` *cond* `.`

[1] A Priced-Timed Maude module stored in the Real-Time Maude database. We use $module_{\texttt{RTM}}$ to represent this module.

[2] All but the last step of this series of transformations is almost identical to those of the function `procTimedCommand`, which handles parsing and transformation of user input in Real-Time Maude. This is discussed in [22]. Also note that `procPriceTimedCommand` actually transforms a 4-tuple: a Priced-Timed Maude command and module along with a timed database and a tick mode. The timed database and tick mode parts are handled exactly like in Real-Time Maude and are therefore not discussed here.

now has the Real-Time Maude representation

```
crl [l] : {S with cost OLDCOST:Cost}
          =>
          {S' with cost OLDCOST:Cost pluss C} if cond .
```

The new `Cost` variable `OLDCOST` is inserted into both sides so that the left-hand side represents a complete state of the form `{`S `with cost` C`}`. This variable also has to be added to the right-hand side as the cost increases with the rule. In addition, if a specification is object-oriented, a variable `C:Configuration` is added to both sides so that the `{_}` captures the whole system state in both sides of the rule. When priced rules are applied to the term representing the state, a new `Cost` term is added to the state's `with cost` element and the global cost is increased. It is essential for the system to be flat in order for cost to be added up correctly. This means that all priced rules must add to the same `with cost` term in the system. If `with cost` terms are allowed to exist on different levels in the system, there is not one definite global cost but many local ones. This can be remedied by, for instance, making a function that scans the system between rule applications and collects `with cost` terms from the different levels. However, implementing such a solution would be too time consuming for this thesis.

A priced *tick* rule

```
crl [tick] : {S} => {S} in time T with cost C with cond .
```

is now transformed into the Real-Time Maude tick rule

```
crl [tick] : {S with cost OLDCOST:Cost}
             =>
             {S' with cost OLDCOST:Cost pluss C} in time T if cond .
```

"Normal" Priced-Timed Maude tick rules are translated in the same manner, by inserting the `Cost` variable `OLDCOST` in both sides of the rule. This is done to ensure the whole state of the system is represented in both sides.

6.2 Transformations for Command Executions

When executing the commands, initial terms and search patterns need to be transformed to be on the same form as the Real-Time Maude representation of the state, to ensure search patterns are reachable from initial terms and initial terms match priced rules. In addition, modules need to be transformed so that all priced rules have the correct form, and if a cost limit is given, all priced rules need extra conditions to satisfy this limit. Finally, atomic propositions should hold whether they contain a `with cost` term or not, this is tackled by the final module transformation.

6.2.1 Module Transformation: `pricifyMod`

The transformation `pricifyMod` is used on modules to transform all priced rules to a standard form. Specifically, a priced rule

(1) `crl [`*label*`] :` t `=>` t' `with cost` C `if` *cond* `.`

is transformed to

(2a) `crl [`*label*`] : {C:Configuration` t `with cost OLDCOST:Cost }`
`=>`
`{C:Configuration` t' `with cost` C `pluss OLDCOST:Cost}` `if` *cond* `.`

if the system is object-oriented; otherwise it is transformed to

(2b) `crl [`*label*`] : {`t `with cost OLDCOST:Cost }`
`=>`
`{`t' `with cost` C `pluss OLDCOST:Cost}` `if` *cond* `.`

The variables `C` and `OLDCOST` are new variables[3]. The variable `C` is inserted only if the system is object-oriented to ensure the whole system is encapsulated by `{_}`. A priced *tick* rule

(3) `crl [`*tick*`] : {`S`} => {`S'`} in time` T `with cost` C `.`

it is transformed to

(4) `crl [`*tick*`] : {`S `with cost OLDCOST:Cost }`
`=>`
`{`S' `with cost` C `pluss OLDCOST:Cost} in time` T `if` *cond* `.`

for both object-oriented and non-object-oriented modules.

In what follows, I present the Maude definition of the transformation for conditional rules.

The function `withCostTerm` determines whether a term contains a `with cost` term. This function searches for the `_with cost_` operator, when one is found it makes sure that it forms a proper `PricedSystem` or `PricedTimedSystem` term:

```
op withCostTerm : Module Term  -> Bool .
op withCostTerm : TermList  -> Bool .
eq withCostTerm(M, T) =
   withCostTerm(T) and
   (leastSort(M, T) == 'PricedSystem or leastSort(M, T) == 'PricedTimedSystem) .
eq withCostTerm(F[TL])
   = if F == '_with‘cost_ then true else withCostTerm(TL) fi .
ceq withCostTerm((T, TL)) = withCostTerm(T) or withCostTerm(TL)
 if TL =/= empty .
eq withCostTerm(T) = false [owise] .
```

The function `pricedRule` determines whether a rule is a priced rule or not:

```
op pricedRule : Rule -> Bool .
eq pricedRule(crl LHS => RHS if COND [AS] .) = pricedRule(rl LHS => RHS [AS] .) .
eq pricedRule(rl LHS => RHS [AS] .) = withCostTerm(RHS) .
```

The function `costPart` extracts the cost part of a term `T`. For instance, if we have a term t `with cost` C where C is the cost expression of the term, only the cost expression is extracted:

[3] If variables by these exact names already exist in the rule, variables with names `C` or `OLDCOST` suffixed by `_`N where N is a natural number is inserted instead.

```
op costPart : NoTerm -> NoTerm .
op costPart : Module Term -> Term .
eq costPart(M, T) = costPart(getTerm(metaReduce(M, T))) .
op costPart : Term -> Term .
eq costPart('_in`time_[T, T']) = costPart(T) .
eq costPart(''{_`}[T]) = costPart(T) .
eq costPart('_with`cost_[T,T']) = T' .
eq costPart(NTerm) = 'infcost.CostInf .
```

The following shows the auxiliary function `newCostVar` used by the transformation, this function is based on the Real-Time Maude function `myNewVar`. This function searches through a term to make sure a variable with a given name can be added safely, before returning the full name of the variable:

```
op newCostVar : Term Qid Nat -> Variable .
ceq newCostVar(T, Q) = if Q' in vars(T) then newCostVar(T, Q, 1) else Q' fi
 if Q' := conc(Q, ':Cost) .
ceq newCostVar(T, Q, N) = if Q' in vars(T) then newCostVar(T, Q, N + 1) else Q' fi
 if Q' := conc(index(conc(Q, '#), N), ':Cost) .
```

The last auxiliary function I show is `pricifyTerm`. This function adds a term `with cost OLDCOST:Cost` to a term. In addition, the term is encapsulated in the `{_}` operator if this is not done already. Furthermore, if the sort `SystemState` is a subsort of the sort `Configuration`, i.e., the system is object-oriented, and the term was not encapsulated in `{_}`, then a variable `C:Configutation` is also added into the term. This means that, e.g., left-hand sides of priced instantaneous rules t are rewritten to `{C:Configuration` t `with cost OLDCOST:Cost}` while those of a priced tick rule with the form `{`S`}` are rewritten to `{`S `with cost OLDCOST:Cost}`.

```
op pricifyTerm : Bool Term Module -> Term .
eq pricifyTerm(true, T, M)  = makePriced(T, newCostVar(T, 'OLDCOST)) .
ceq pricifyTerm(false, T, M)
    = if OO then
       if withCostTerm(T) then
        makePriced(makePriced(''{_`}['__[Q',stripCost(T)]], Q), costPart(T))
       else
        makePriced(''{_`}['__[Q',T]], Q)
       fi
      else
       makePriced(''{_`}[T], Q)
      fi
 if Q := newCostVar(T, 'OLDCOST)
    /\ OO := subsortOf('Configuration, 'SystemState, M)
    /\ Q' := if OO then newConfVar(T, 'C) else 'none fi [owise] .
```

The function `pricifyMod` starts by extracting the rule set `RLS` from the module before passing it to the function `pricifyRls`:

```
op pricifyMod : Module -> Module .
eq pricifyMod(FM:FModule) = FM:FModule .
eq pricifyMod(mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm)
  = (mod H is IL sorts SS . SSDS OPDS MAS EQS
     pricifyRls(RLS, mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm) endm) .
```

The function `pricifyRls` examines each rule `RL` to determine whether it is a priced rule, tick rule or neither. Only priced rules and tick rules are passed on to the function `pricifyRule` that does the actual transformation:

```
op pricifyRls : RuleSet Module -> RuleSet .
eq pricifyRls(RL RLS, M) = if pricedRule(RL) or tickRule(RL) then
                           pricifyRule(RL, M) else RL fi
    pricifyRls(RLS, M) .
eq pricifyRls(none, M) = none .
```

The following code uses `pricifyTerm` to rewrite both sides of instantaneous rules of the form shown as (1) to the form (2*a*) if the specification is object-oriented, otherwise (2*b*):

```
ceq pricifyRule(crl LHS => '_with`cost_[RHS,T] if COND [AS] ., M) =
     (crl pricifyTerm(false, LHS, M)
      =>
      pricifyTerm(false, '_with`cost_[RHS,T], M) if COND [AS] .)
 if not globalSystemTerm(LHS) .
```

The following code uses the function `pricifyTerm` to rewrite each tick rule of the form (3) to the form (4):

```
eq pricifyRule(crl ''{_`}[LHS]
                   => '_with`cost_['_in`time_[''{_`}[RHS],T],T'] if COND [AS] ., M) =
   (crl pricifyTerm(true, ''{_`}[LHS], M)
    =>
    pricifyTerm(true, '_in`time_[''{_`}['_with`cost_[RHS,T']],T], M)
    if COND [AS] .) .
```

The following example illustrates exactly what effect the `pricifyMod` transformation has on a module.

Example 6.2.1 *After performing the* `pricifyMod` *transformation on the module* `PRICED-TIMED-OO-LIGHT-SWITCH` *from Section 4.5.2 the priced rules in this module look as follows:*

```
rl [turn-on] : {C:Configuration < O : Switch | wattage : W, timer : INF,
                                               status : off >
                                               with cost OLDCOST:Cost}
               =>
               {C:Configuration < O : Switch | timer : 5, status : on >
                with cost W pluss OLDCOST:Cost} .

crl [tick] : {S with cost OLDCOST:Cost}
             =>
             {delta(S, R) with cost ((rate(S) * R) pluss OLDCOST:Cost)}
             in time R if R <= mte(S) [nonexec] .
```

6.2.2 Module Transformation: `costlimitMod`

When commands with a cost limit are executed, the module is transformed so that no states with cost exceeding this limit are explored. This is accomplished by the transformation `costlimitMod`, which ensures that priced rules are not executed if this would make the global cost in the system exceed a given cost limit. The transformation adds a condition that requires that the cost part of the right-hand side

of all priced rules not to exceed the given limit. This transformation assumes `pricifyMod` has already been performed on the module.

Two auxiliary functions `cheaperCond` and `cheaperEqCond` are defined to help generate new cost bound conditions in the right-hand side of the rules. The first term `T` is the left-hand side and `T'` is the right-hand side of the comparison operator:

```
ops cheaperCond cheaperEqCond : Term Term -> Condition .
eq  cheaperCond(T,T') = 'true.Bool = '_cheaper‘than_[T,T'] .
eq  cheaperEqCond(T,T')
    = 'true.Bool = '_or_['_cheaper‘than_[T,T'],'_==_[T,T']] .
```

The transformation is initiated by the function `costLimitMod` that extracts the rule set `RLS` from the module:

```
op costLimitMod : Module Bool Term ->  Module .
eq costLimitMod(FM:FModule, B, T) = FM:FModule .
eq costLimitMod(mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm, B, T)
   = (mod H is IL sorts SS . SSDS OPDS MAS EQS costLimitRls(RLS, B, T) endm) .
```

The function `costLimitRls` examines each rule `RL` in the rule set and uses the function `pricedRule` to determine whether the rule is a priced rule or not. Only priced rules are affected by the transformation:

```
op costLimitRls : RuleSet Bool Term -> RuleSet .
eq costLimitRls(RL RLS, B, T) = if pricedRule(RL) then
                                   costLimitRule(RL, B, T) else RL fi
                                   costLimitRls(RLS, B, T) .
eq costLimitRls(none, B, T) = none .
```

When a priced rule is encountered, an additional condition `COND'` is appended to the existing condition `COND`. If the boolean `B` is true, the condition C `cheaper than T` (where C is the cost part of the right-hand side) is added using `cheaperCond`. Otherwise, the condition
C `cheaper than or eq T` is added using `cheaperEqCond`:

```
op costLimitRule : Rule Bool Term -> Rule .
ceq costLimitRule(crl LHS => RHS if COND [AS] ., B, T)
    = (crl LHS => RHS if COND /\ COND' [AS] .)
 if COND' := if B then cheaperCond(costPart(RHS), T)
             else cheaperEqCond(costPart(RHS), T) fi .
```

The function transforms unconditional rules in the same way, making them conditional and adding the appropriate condition:

```
ceq costLimitRule(rl LHS => RHS [AS] ., B, T)
    = (crl LHS => RHS if COND [AS] .)
 if COND := if B then cheaperCond(costPart(RHS), T)
            else cheaperEqCond(costPart(RHS), T) fi .
```

6.2.3 Module Transformation: `pricifyProperties`

For temporal logic model checking, if an atomic proposition p is defined by the user so that {s} |= p is true, then {s `with cost` c} |= p must also hold for the internal representation for any cost c. That is achieved by adding the following conditional equation to the definition of the satisfaction relation:

```
ceq {S:SystemState with cost C:Cost} |= P:Prop = true.Bool
 if {S:SystemState} |= P:Prop = true.Bool .
```

The following function `pricifyProperties` inserts this equation into a specification:

```
op pricifyProperties : Module -> Module .

eq pricifyProperties(mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm) =
                    (mod H is IL sorts SS . SSDS OPDS MAS
                    (EQS (ceq '_|=_['`{_`}['_with`cost_['S:SystemState, 'C:Cost]],
                                 'P:Prop] = 'true.Bool
                    if '_|=_['`{_`}['S:SystemState], 'P:Prop] = 'true.Bool [none] . ))
                    RLS endm) .
```

6.2.4 Transforming the Initial Term: `pricifyInit`

The initial state {S} given by the user must be transformed to a Real-Time Maude initial state {S `with cost free`} when executing a command. This transformation is done by the following function `pricifyInit`:

```
op pricifyInit : Term -> Term .
eq pricifyInit(T) = makePriced(T, 'free.Cost) .
```

The transformation uses only one auxiliary function `makePriced`. This function adds a given cost to a term, if the term has no `_with cost_` operator present, one is inserted:

```
op makePriced : Term Term -> Term .
eq makePriced('_in`time_[T, T'], T'')
   = '_in`time_[makePriced(T, T''), T'] .
eq makePriced('`{_`}[T], T') =  '`{_`}[makePriced(T, T')] .
eq makePriced('_with`cost_[T,T''], T')
= '_with`cost_[T, '_pluss_[T'',T']] .
eq makePriced(T, T') = '_with`cost_[T,T'] [owise] .
```

6.2.5 Transforming Search Patterns: `pricifyPattern`

A transformation called `pricifyPattern` adds new cost variable `TOTAL_COST_INCURRED` into search patterns, i.e., search patterns on the form {S'} are rewritten to {S' `with cost TOTAL_COST_INCURRED:Cost`}. This ensures that a search pattern matches a system with a `with cost` term and that the convenient `TOTAL_COST_INCURRED` substitution is displayed.

The transformation uses the `makePriced` function in conjunction with the `newCostVar` function to add a new variable `TOTAL_COST_INCURRED:Cost`:

```
op pricifyPattern : Term -> Term .
eq pricifyPattern(T) = makePriced(T, newCostVar(T,'TOTAL_COST_INCURRED)) .
```

6.3 Commands and Algorithms

This section discusses how each command is transformed into a Real-Time Maude command and how the transformations described in Section 6.2 are used. In general, all Priced-Timed Maude commands transform the module with `pricifyMod` and with `costlimitMod` if a price limit is given. Furthermore, all commands use `pricifyInit` to add a zero-value cost to the initial term. Additionally, all the search commands use the `pricifyPattern` transformation on the search pattern. Finally, the transformed module and terms are passed to a Real-Time Maude meta-function that executes the command.

6.3.1 Priced-timed Rewriting: `ptrew` and `ptfrew`

When executing the command `ptrew` with a cost limit, the following transformation takes place:

$$\begin{gathered}(\texttt{ptrew(INITIALSTATE, CHEAPER, COSTBOUND, } Args\texttt{)}, module_{\texttt{RTM}}) \\ \Downarrow \texttt{procPriceTimedCommand} \\ (\texttt{trew((pricifyInit(INITIALSTATE), } Args\texttt{))}, \\ \texttt{costlimitMod(pricifyMod(}module_{\texttt{RTM}}\texttt{), CHEAPER, COSTBOUND))}\end{gathered}$$

We see that the command `ptrew` transforms to the Real-Time Maude command `trew`. The module is transformed, first with `pricifyMod`, then cost limits according to the terms `CHEAPER` and `COSTLIMIT` are added to each priced rule using the `costlimitMod` transformation. Furthermore, the initial state `INITIALSTATE` is transformed using the `pricifyInit` transformation. Other arguments such as time limit and the tick mode are unaffected and are represented by *Args*.

The functions `pricedTimedMetaRewrite` and `pricedTimedMetaFRewrite` are found in the module `PRICED-TIMED-REWRITI` Both commands transform their module and arguments locally then pass these to `timedMetaRewrite` and `timedMetaFRewrite`:

```
op pricedTimedMetaRewrite : Module Term Bound ComparisonOp Term
                            TickMode Bool Term -> ResultPair .
```

The following arguments are used for `ptrew` with a cost limit: `M` is the module to rewrite in; `INITIALSTATE` represents the initial state; `BOUND` is the bound on the number of rewrite steps; `COMP` is the comparison operator on the time limit; `TIMEBOUND` is the time limit; `TM` is the tick mode; `CHEAPER` is a boolean signifying whether cost should be cheaper than or cheaper than or equal to the cost limit; and `COSTBOUND` is the cost limit.

This version of the command takes a cost limit in addition to the time and rewrite step limit. Therefore, the module `M` is first transformed using `pricifyMod` then `costlimitMod` with the appropriate limit. In addition, `with cost free` is added into the initial term before sending the job to Real-Time Maude meta-rewrite function:

```
eq pricedTimedMetaRewrite(M, INITIALSTATE, BOUND, COMP, TIMEBOUND, TM, CHEAPER,
                          COSTBOUND)
   = timedMetaRewrite(costLimitMod(pricifyMod(M), CHEAPER, COSTBOUND),
                      pricifyInit(INITIALSTATE), BOUND, COMP, TIMEBOUND, TM) .
```

The next version of the priced-timed rewrite function does not use a price limit. Therefore, the `costlimitMod` transformation is not applied to the module before the job is sent to the appropriate Real-Time Maude meta-rewrite function:

```
  op pricedTimedMetaRewrite : Module Term Bound ComparisonOp Term TickMode -> ResultPair .

  eq pricedTimedMetaRewrite(M, INITIALSTATE, BOUND, COMP, TIMEBOUND, TM)
     = timedMetaRewrite(pricifyMod(M), pricifyInit(INITIALSTATE), BOUND, COMP,
                        TIMEBOUND, TM) .
```

The code for `pricedTimedMetaRewrite` and `pricedTimedMetaFRewrite` is almost the same, with the difference being `pricedTimedMetaRewrite` uses `timedMetaRewrite`.

6.3.2 Priced-Timed Search: `ptsearch`

When executing the command `ptsearch` with a cost limit, the following transformation takes place:

$$\begin{array}{c}(\texttt{ptsearch}(\texttt{INITIALSTATE}, \texttt{SEARCHPATTERN}, \texttt{CHEAPER}, \texttt{COSTBOUND}, \mathit{Args}), \mathit{module}_{\texttt{RTM}}) \\ \Downarrow \texttt{procPriceTimedCommand} \\ \texttt{tsearch}(\texttt{pricifyInit}(\texttt{INITIALSTATE}), \texttt{pricifyPattern}(\texttt{SEARCHPATTERN}), \mathit{Args}), \\ \texttt{costlimitMod}(\texttt{pricifyMod}(\mathit{module}_{\texttt{RTM}}), \texttt{CHEAPER}, \texttt{COSTBOUND}))\end{array}$$

Priced-timed search is defined by the function `pricedTimedSearch` in the module `PRICED-TIMED-SEARCH`. This function performs all necessary transformations on the module, initial term, and search pattern. The function `pricedTimedSearch` effectively transforms a call to `ptsearch` into a call to the function `timedMetaSearch` that implements Real-Time Maude's `tsearch` command. The following shows the code for the priced-timed search command:

```
  op pricedTimedSearch : Module Term Term Condition Qid Bound
                         Nat ComparisonOp Term TickMode Bool Term
                         -> ResultTriple? .
```

The argument `SEARCHPATTERN` represents the search pattern; `COND` is the condition on the search pattern; `Q` is the arrow (`*`, `!`, etc); `D` is the minimum depth; and `N` the number of solutions to find. The following version of the priced-timed search has a time limit so both the `pricifyMod` and `costlimitMod` transformations are performed:

```
  eq pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN,
                       COND, Q, D, N, COMP, TIMEBOUND, TM, CHEAPER, COSTBOUND)
     = timedMetaSearch(costLimitMod(pricifyMod(M), CHEAPER, COSTBOUND),
                       pricifyInit(INITIALSTATE), pricifyPattern(SEARCHPATTERN),
                       COND, Q, D, N, COMP, TIMEBOUND, TM) .
```

The second version of the command has no price limit, therefore, the `costlimitMod` transformation is not performed:

```
  op pricedTimedSearch : Module Term Term Condition Qid Bound
                         Nat ComparisonOp Term TickMode -> ResultTriple? .

  eq pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN,
                       COND, Q, D, N, COMP, TIMEBOUND, TM)
     = timedMetaSearch(pricifyMod(M), pricifyInit(INITIALSTATE),
                       pricifyPattern(SEARCHPATTERN),
                       COND, Q, D, N, COMP, TIMEBOUND, TM) .
endfm
```

6.3.3 Find Cheapest

For each iteration of the `find cheapest` command a call to `ptsearch` is made with a stricter cost limit. The second step of the following transformations is taking place for *each iteration*:

$$\begin{array}{c}(\texttt{find cheapest}((\texttt{INITIALSTATE}, \texttt{SEARCHPATTERN}, \texttt{CHEAPER}, \texttt{COSTBOUND}, Args), module_{\texttt{RTM}}) \\ \Downarrow \texttt{procPriceTimedCommand} \\ (\texttt{ptsearch}(\texttt{INITIALSTATE}, \texttt{SEARCHPATTERN}, \texttt{CHEAPER}, \texttt{COSTBOUND}, Args), module_{\texttt{RTM}}) \\ \Downarrow \texttt{pricedTimedSearch} \\ (\texttt{tsearch}(\texttt{pricifyInit}(\texttt{INITIALSTATE}), \texttt{pricifyPattern}(\texttt{SEARCHPATTERN}), Args), \\ \texttt{costlimitMod}(\texttt{pricifyMod}(module_{\texttt{RTM}}), \texttt{CHEAPER}, \texttt{COSTBOUND}))\end{array}$$

The code for the `find cheapest` command is defined by the function `findCheapest` found in the module `FIND-CHEAPEST`. All transformations of the module, initial term, and search pattern are performed by `pricedTimedSearch`. The algorithm employed by the function `findCheapest` obtains an initial solution by using `pricedTimedSearch` then stores this solution and, thereafter, uses `pricedTimedSearch` with a cost limit cheaper than the cost of the current solution. This is repeated until there are no more solutions. Therefore, the current solution is the optimal one.

More complex algorithms for computing the cheapest states such as linear programming combined with branch and bound were considered, but due to the flexibility of Maude's rewrite rules, too many properties about a specification are undecidable. Therefore, a branch and bound algorithm would end up having bad bounding functions and search a large part of the state space. We opt to use Real-Time Maude's built-in meta-search because of the aforementioned reason and the fact that during tests of different implementations, the built-in meta-search proved itself as more efficient.

```
op findCheapest : Module Term Term Condition Qid Bound
                  ComparisonOp Term TickMode -> ResultTriple? .

op findCheapest : Module Term Term Condition Qid Bound
                  ComparisonOp Term TickMode Term ResultTriple?
                  -> ResultTriple? .
```

The initial solution is obtained by sending the arguments provided to the `find cheapest` command to the function `pricedTimedSearch` and storing the result in `THIS_SEARCH`. If the first result was not a failure, `findCheapest` is called again. This time with a cost limit set to cheaper than the cost of the current result:

```
ceq findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP, TIMEBOUND, TM)
    = if THIS_SEARCH =/= failure then
       findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP, TIMEBOUND, TM,
                    costPart(getTerm(THIS_SEARCH)), THIS_SEARCH)
      else
       failure
      fi
if THIS_SEARCH :=
   pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                     0, COMP, TIMEBOUND, TM, true, 'infcost.CostInf) .
```

The `findCheapest` function calls itself repeatedly passing its arguments to the `pricedTimedSearch` function as long as it finds cheaper solutions. Once there are no more solutions, the solution stored in `PREV_SEARCH`, the optimal solution is returned:

```
ceq findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                 TIMEBOUND, TM, COSTBOUND, PREV_SEARCH)
    = if THIS_SEARCH =/= failure then
      findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP, TIMEBOUND, TM,
                   costPart(getTerm(THIS_SEARCH)), THIS_SEARCH)
      else
       PREV_SEARCH
      fi
if THIS_SEARCH :=
   pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                     0, COMP, TIMEBOUND, TM, true, COSTBOUND) .
```

6.3.4 Binary Find Cheapest

For each iteration in the `binary find cheapest` command a call to `ptsearch` is made:

$$
\begin{array}{c}
(\texttt{binary find cheapest(INITIALSTATE,SEARCHPATTERN,} \\
\texttt{CHEAPER,COSTBOUND,}\mathit{Args}\texttt{)},\mathit{module}_{\texttt{RTM}}) \\
\Downarrow \texttt{procPriceTimedCommand} \\
(\texttt{ptsearch(INITIALSTATE,SEARCHPATTERN,CHEAPER,COSTBOUND,}\mathit{Args}\texttt{)},\mathit{module}_{\texttt{RTM}}) \\
\Downarrow \texttt{pricedTimedSearch} \\
(\texttt{tsearch(pricifyInit(INITIALSTATE),pricifyPattern(SEARCHPATTERN)},\mathit{Args}\texttt{)}, \\
\texttt{costlimitMod(pricifyMod(}\mathit{module}_{\texttt{RTM}}\texttt{),CHEAPER,COSTBOUND)})
\end{array}
$$

The binary version of the `find cheapest` command uses the function `findCheapestBin` that is located in the module `FIND-CHEAPEST-BINARY`. This function is similar to `findCheapest` except in how it applies the `costlimitMod` transformation in a binary search fashion to obtain the next better solution. An initial solution is obtained by performing a `ptsearch` with no cost limit for a matching state. Once a solution is found, it is stored and a new search is made for a solution with half or less cost than the previous one. If a solution is not found, the algorithm tries to find a solution between half the cost of the current best and the current best. This is repeated in a standard binary search pattern, until there are no more states to search and an optimal solution is found.

The function `metaDiv2` is used to divide a meta cost terms by 2. This is the only function that uses the `div2` operator defined as part of the cost domain:

```
op metaDiv2 : Module Term -> Term .
eq metaDiv2(M, T) = getTerm(metaReduce(M,'div2[T])) .
```

The function `metaAdd` is used to add two meta costs, while `metaAvg` is used to take the average of two given meta costs:

```
ops metaAdd metaAvg : Module Term Term -> Term .
eq metaAdd(M, T, T') = getTerm(metaReduce(M,'_pluss_[T,T'])) .
eq metaAvg(M, T, T') = metaDiv2(M, metaAdd(M, T, T')) .

op findCheapestBin : Module Term Term Condition Qid Bound
                     ComparisonOp Term TickMode -> ResultTriple? .

op findCheapestBin : Module Term Term Condition Qid Bound
                     ComparisonOp Term TickMode Term ResultTriple? -> ResultTriple? .
```

The initial call to the `findCheapestBin` function runs in the same way as the `findCheapest` function apart from one thing – the cost limit `COSTLIMIT` that is being passed to the next iteration of the function is half the value of the cost of the current solution:

```
  ceq findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP, TIMEBOUND, TM)
      = if THIS_SEARCH =/= failure then
         findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                         TIMEBOUND, TM,
                         metaAvg(M, costPart(getTerm(THIS_SEARCH)), 'free.Cost),
                         THIS_SEARCH)
        else
         failure
        fi
  if THIS_SEARCH := pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                                      0, COMP, TIMEBOUND, TM, true, 'infcost.CostInf) .
```

Subsequent calls to `findCheapestBin` proceed by using the `metaAdd` and `metaAvg` functions to search for better solutions in a binary search fashion:

```
  ceq findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                      TIMEBOUND, TM, COSTBOUND, PREV_SEARCH)
      = if THIS_SEARCH =/= failure then
         findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                         TIMEBOUND, TM,
                         metaAvg(M, costPart(getTerm(THIS_SEARCH)), 'free.Cost),
                         THIS_SEARCH)
        else
         if CURRENT == COSTBOUND then
          PREV_SEARCH
         else
          findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                          TIMEBOUND, TM, CURRENT, PREV_SEARCH)
         fi
        fi
  if CURRENT := metaAvg(M,  costPart(getTerm(PREV_SEARCH)), COSTBOUND)
     /\
     THIS_SEARCH := pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                                      0, COMP, TIMEBOUND, TM, true, COSTBOUND) .
endfm
```

6.3.5 Priced Find Earliest

When the `priced find earliest` command is called with a price limit, the following transformation takes place:

$$\begin{array}{c}(\texttt{priced find earliest(INITIALSTATE,SEARCHPATTERN,}\\ \texttt{CHEAPER,COSTBOUND,}\mathit{Args}\texttt{)},\mathit{module}_{\texttt{RTM}})\\ \Downarrow \texttt{procPriceTimedCommand}\\ (\texttt{find earliest(pricifyInit(INITIALSTATE),pricifyPattern(SEARCHPATTERN)},\mathit{Args}\texttt{)},\\ \texttt{costlimitMod(pricifyMod(}\mathit{module}_{\texttt{RTM}}\texttt{),CHEAPER,COSTBOUND))}\end{array}$$

There is no code for this command as the transformed input is simply sent to the Real-Time Maude function `findEarliest` by part of the preprocessor and the result is displayed on screen.

6.3.6 Timed Model Checking

When the `pmc` command is called, the following transformation takes place:

$$(\texttt{pmc(INITIALSTATE}, Args\texttt{)}, module_{\texttt{RTM}})$$
$$\Downarrow \texttt{procPriceTimedCommand}$$
$$(\texttt{mc(pricifyInit(INITIALSTATE)}, Args\texttt{)}, \texttt{pricifyProperties(pricifyMod(}module_{\texttt{RTM}}\texttt{))})$$

No specific model checking code apart from the function `pricifyProperties` is added by Priced-Timed Maude. After this transformation, the model checker is invoked with the transformed module and initial state.

6.4 Defining the Syntax of Priced-Timed Maude

This section shows how the user-level syntax of modules and commands is defined in Priced-Timed Maude. Parsing issues are not covered in this thesis as methods identical to Real-Time Maude's are used when parsing user input and preprocessing commands; this is discussed in [22].

Module syntax is specified the same way as in Real-Time Maude. User-level syntax for Priced-Timed Maude's modules is given in the following module, which extends the module `REAL-TIME-MAUDE-SYNTAX` that defines the user-level syntax of Real-Time Maude:

```
fmod PRICED-MODULE-SYNTAX is
 including REAL-TIME-MAUDE-SYNTAX .

 op pmod_is_endpm : @Interface@ @SDeclList@ -> @Module@ .
 op pomod_is_endpom : @Interface@ @SDeclList@ -> @Module@ .

 op ptmod_is_endptm : @Interface@ @SDeclList@ -> @Module@ .
 op ptomod_is_endptom : @Interface@ @ODeclList@ -> @Module@ .
endfm
```

The user-level syntax for Priced-Timed Maude commands is defined as follows. Because of numerous instances of similar code, only parts are shown, the omitted ones are denoted by "...". The complete list is found in Appendix B:

The following shows how the meta-level syntax of some the Priced-Timed Maude commands are defined:

```
 ...
 op ptsearch_=>*_in time <_with cost <_. : @Bubble@ @Bubble@ @Bubble@ @Bubble@
                                          -> @Command@ .
 op ptsearch_=>*_in time <_with cost <=_. : @Bubble@ @Bubble@ @Bubble@ @Bubble@
                                           -> @Command@ .
 op ptsearch_=>*_in time <=_with cost <_. : @Bubble@ @Bubble@ @Bubble@ @Bubble@
                                           -> @Command@ .
 op ptsearch_=>*_in time <=_with cost <=_. : @Bubble@ @Bubble@ @Bubble@ @Bubble@
                                            -> @Command@ .
 op ptsearch_=>!_in time <_with cost <_. : @Bubble@ @Bubble@ @Bubble@ @Bubble@
                                          -> @Command@ .
 op ptsearch_=>*_with no time limit with cost <_. : @Bubble@ @Bubble@ @Bubble@
                                                    -> @Command@ .
 op ptsearch_=>!_with no time limit with cost <_. : @Bubble@ @Bubble@ @Bubble@
                                                    -> @Command@ .
```

```
op ptsearch_=>*_in time <_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                                -> @Command@ .
op ptsearch_=>*_in time <=_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                                 -> @Command@ .
op ptsearch_=>!_in time <_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                                -> @Command@ .
op ptsearch_=>*_with no limits . : @Bubble@ @Bubble@ -> @Command@ .
op ptsearch_=>!_with no limits . : @Bubble@ @Bubble@ -> @Command@ .
...
op find cheapest_=>*_in time <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
op find cheapest_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
op find cheapest_=>*_with no time limit. : @Bubble@ @Bubble@ -> @Command@ .

op binary find cheapest_=>*_in time <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
...
op ptfrew_in time <=_with cost <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
op ptfrew_in time <=_with cost <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
op ptfrew_in time <=_with no cost limit. : @Bubble@ @Bubble@ -> @Command@ .
op ptfrew_with no time limit with cost <_. : @Bubble@ @Bubble@ -> @Command@ .
op ptfrew_with no limits. : @Bubble@ -> @Command@ .
...
op ptrew_in time <=_with cost <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
...
op priced find earliest_=>*_with no cost limit. : @Bubble@ @Bubble@ -> @Command@ .
op priced find earliest_=>*_with cost <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
op priced find earliest_=>*_with cost <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
...
```

The sort `@Bubble@` is used for the commands' arguments. We recall the syntax of the `ptsearch` command given in the previous chapter:

`(ptsearch` [`[`n`]`] *initialState* `=>*` *searchPattern* [`such that` *cond*]
`in time` $\trianglelefteq$ T `with cost` $\trianglelefteq$ C `.)`

Two parameters precede the `=>`; one of these is optional but both are represented by the same `@Bubble@`. When parsing the command, a function resolving these bubbles determine what part of the arguments are present and resolve abbreviations (such as initial states defined as constants) to terms. Also note that the operators `<` and `<=` are part of the actual command rather than parameters, this is to minimize the amount of parsing needed later on. The same applies to the arrows `=>*` and `=>!`.

Example 6.4.1 *Section 4.5.2 introduced the module* `PRICED-TIMED-OO-LIGHT-SWITCH` *and an initial state* `init1` *with one switch in the module* `TEST-TWO-LIGHTS`*. The following is a search in that system*

```
(ptsearch [1] init1 =>* {< "Driveway" : Switch | timer : 4 >} in time <= 1
                        with cost <= 50 .)
```

where the first bubble is '`[1] init1`'; the second bubble is '`{< "Driveway" : Switch | timer : 4 >}`'; third bubble is the time limit `1`; and the fourth bubble is the cost limit `50`. Here we see that the first bubble does not represent one but two parameters, both the amount of solutions wanted and the initial state, which in this case is the operator `init1` that needs to be resolved by the bubble solve function.

Chapter 7

Case Studies

In this chapter, we look at 3 priced-timed systems: the *airplane landing problem* (ALP) [12], *energy task graph scheduling* (ETGS) [13], and the *subway passenger routing* (SPR) problem and show how to model and analyze them using Priced-Timed Maude. The first two systems are common benchmarks and are modeled using Priced-Timed Automata (PTA) [13] using the tool Uppaal CORA [10] in the article [13] and on the tool's webpage. The first 2 systems are provided to show how Uppaal CORA and Priced-Timed Maude differ in specification style and performance (both aspects are extensively discussed in Chapter 8). The last system is given to illustrate the flexibility of modeling priced-timed systems using Priced-Timed Maude.

7.1 The Airplane Landing Problem

In the *airplane landing problem* (ALP) [12], aircraft landings are scheduled within a given time window onto a set of runways. If an aircraft is assigned a landing time that deviates from a given target time, then it has to accelerate or hold in the air. This leads to using more fuel than planned, and additional cost is incurred. The objective is to minimize this cost.

This problem was chosen because ALP is a often cited in papers discussing priced-timed systems.

7.1.1 The Problem

The task is to assign landing times to a set of aircraft, given a set of runways, while minimizing the number of planes deviating from their original, or *target* schedules.

Each plane is given an earliest, target, and latest landing time. Any cost incurred in the system is due to deviation from its target landing time. In addition, the aircraft are further classified according type, e.g. Boeing 747 or Airbus A320. The reason for this classification is that different plane sizes generate different amounts of turbulence wake when landing. This is relevant to landing separation times.

When a plane comes within range of the airport, the air traffic control tower will assign it a landing time and an available runway before it runs out of fuel. When determining landing schedules, we have to take into account that only one plane may land on a particular runway on any given time and appropriate time gaps have to be provided between landing aircraft due to the turbulence wake created as a plane taxies down. The required separation time depends on the type of aircraft involved. For instance, a 747 can both create and handle more turbulence than a smaller plane, such as a four-seater, single-engine Cessna.

Before a landing time can be assigned, a plane is assumed to arrive at the target time, flying at a cruising speed that has been determined as the most fuel efficient for that aircraft. Therefore, a landing based on its target time will not add any cost to the system. However, if a plane is assigned a landing time earlier than its estimated target time, it must increase its speed, thereby consuming more fuel. This is refereed to as a plane's *early rate*. The increase in fuel consumption will be at its maximum (cost is equal to the value of the early rate) if the plane is assigned the earliest possible landing time but will decrease linearly towards 0 for landing times closer to its target time If a plane is assigned a landing time past its target time, it must stay in the air longer than expected. This also leads to an increased fuel consumption, referred to as the plane's *late rate*. The plane will add cost to the system that is equal to this rate per time unit. In addition to the early and late fuel consumption rates, there is a one-time penalty associated with a plane landing late.

This type of problem occurs on a large scale at busy airports where making optimal use of a bottleneck resources the runways is crucial for the airport's safe and smooth operation.

7.1.2 Examples

The following example defines an initial state with 3 planes and a runway:

Example 7.1.1 *We define an initial state with 3 planes denoted* `p1`*,* `p2`*, and* `p3`*:*

- `p1` *is a small plane that is set to land at time 9, no earlier than 5, but no later than 12. Further, it has an early rate of 15, late rate of 2 and a one time late penalty of 3.*
- `p2` *is another small plane that has a target and earliest landing time of 5 and a latest landing time 10. Moreover, this plane has an early rate of 25, late rate of 5 and a late penalty of 10.*
- `p3` *is a medium-sized plane that is set to land at time 7, no earlier than 5, and no later than 15. In addition, it has an early rate of 20, a late rate of 3 and a late penalty of 5.*

Furthermore, all three planes have earliest landing time 5. Finally, we have an empty runway `rw1` *for these planes to land on.*

From this initial setup we can generate many schedules, but we will consider only 2 of them: the earliest and the cheapest.

The earliest schedule is obtained in the following example:

Example 7.1.2 *While it is possible for these 3 planes to be grounded in 7 time units, this may not be the cheapest way to do so. Consider the following 2 schedules: first a schedule denoted* S_1*:* `p1` *lands at time 5 with an added cost of 15 because the plane had to accelerate to its maximum airspeed. Now* `p2` *may land at time 6 due to the fact that a small-sized plane can land 1 time unit after another small plane. This will incur a cost of* $5 + 10 = 15$*, one unit of time at the late rate plus the one time late penalty. Finally,* `p3` *may land at time 7: as medium-sized planes have to wait 1 time unit to land after a small-sized aircraft. This is the plane's target landing time, therefore, no extra cost is incurred. The total cost of this schedule is 30.*

Our final example shows us that the earliest schedule is not necessarily the cheapest:

Example 7.1.3 *The second schedule* S_2 *will assign all the planes to land on their target times as follows:*

`p2` *lands at time 5 with no added cost. Likewise,* `p3` *lands at time 7. and finally* `p1` *lands at time 9, also on target. This schedule finishes landing all the planes by time 9 at no cost since all the aircraft landed on their target times and stayed at their advised, cruise speeds.*

7.1.3 Modeling ALP in Priced-Timed Maude

This section provides a method for modeling the ALP problem using Priced-Timed Maude. First we need a representation of planes, which we model with instances of the class `Plane`, with the following attributes:

- `type`: the type of a plane, for this example, we will rely on a generic type: `small`, `big`, and `medium`.
- `earliest`: the earliest possible landing time.
- `target`: the time a plane can land without incurring any extra cost.
- `latest`: the latest possible time a plane can land before it runs out of fuel and a disaster occurs.
- `early`: a cost associated when a plane increases airspeed in order to land earlier than its target time. The full value of the attribute will be incurred if the plane has to land at its earliest time (after which it deceases linearly until it reaches 0 at the target time).
- `late`: the rate at which cost is incurred when a plane has to land beyond its target time.
- `latePenalty`: a one-time penalty incurred for landing beyond its target time.
- `landAt`: identifies the runway the plane has been assigned to. This starts out empty with the special value `noOid`. When a plane has this value as its `landAt` attribute we will refer to it as *unassigned*. Otherwise. it is referred to as *assigned*.
- `landingTime`: this is the time that has been assigned for the plane to land. This starts out the same as `target`, as it is the assumed landing time.
- `clock`: the internal clock of a plane. This starts out at 0.
- `rate`: this is the current rate at which a plane generates cost when it is assigned to an early landing. This rate is based on the time of assignment, earliest landing time, target landing time, and the early rate. For instance, a plane with earliest landing time 2 and target 6 that is being informed at time 3 that it is to land at time 4 will have to accelerate more than if the plane was informed of the same landing time at time 2.

For the class `Runway`, that models runways, we only need 3 attributes:

- `landed`: a list of pairs of the form `(Oid, Time)` where `Oid` identifies a plane and `Time` indicates the time a plane lands.
- `type`: signifies the type of the previously assigned planes, i.e. if a plane of type `big` is assigned a landing time, this attribute becomes `big` so that the system can determine how much separation time is needed for the next assignment. This starts out as the special value `None` which represents an empty runway (, i.e., no planes have been cleared for landing).
- `prevStart`: the time assigned to the previous (last) plane's landing.

We will use the following strategy for solving the problem:

- At system start, all planes are assumed to land on target time.
- An aircraft may be assigned a landing time and a runway number during any period from the time the system starts until its latest landing time.
- When assigning a landing time and runway to an aircraft we will either assign the plane to its target time or we will assign it to the runway's next free time slot. When assigning a plane to the next free slot it may coincide with the plane's target time, it may be earlier than the plane's target, or it may be later.
  - When a plane is assigned a time earlier than its target an early rate is calculated. This rate may be different according to when the plane was informed of the assigned landing time.
  - A plane assigned to its target time will never incur cost.
  - A plane assigned to a time after its target time will incur cost according to its late rate once time is past its target time.

  Clearly, this covers all of a plane's possible landing scenarios.
- When planes have been assigned to a landing time and runway they will continue to cruise at a certain altitude until their designated landing time when they will have to land immediately.

The following object-oriented Priced-Timed Maude specification models the airplane landing problem:

```
(omod DEF-OID is
  sort DefOid .
  op noOid : -> DefOid [ctor] .
  subsort Oid < DefOid .
endom)
```

The included module `DEF-OID` defines the sort `DefOid` that is used for the `noOid` value for unassigned planes.

```
(tomod OID-TIME is
  sort OidTimePair .
  op _‘,_ : Oid Time -> OidTimePair [ctor] .
endtom)

(view OidTime from TRIV to OID-TIME is
  sort Elt to OidTimePair .
endv)
```

The module `OID-TIME-LIST` provides us with a datatype for sets of the sort `OidTimePair` which has the form `(Oid, Time)` and is used in the `landed` attribute of runway to log landings.

```
(tomod OID-TIME-LIST is
  protecting LIST{OidTime} * (sort List{OidTime} to OidTimeList,
                              sort NeList{OidTime} to NeOidTimeList) .
endtom)

(ptomod ALP is
  protecting DEF-OID .
  protecting OID-TIME-LIST .
  protecting POSRAT-COST-DOMAIN .
  protecting NAT-TIME-DOMAIN-WITH-INF .
```

POSRAT-COST-DOMAIN and NAT-TIME-DOMAIN-WITH-INF set cost to run over the positive rational numbers and time to run over the natural numbers, respectively. Cost needs to be positive, rational numbers in this system because of the early rate. This is calculated based on the plane's assigned landing time relative to its earliest landing time and can easily become a rational number.

```
  sorts PlaneType PlaneTypeNone .
  subsort PlaneType < PlaneTypeNone .

  ops big small medium : -> PlaneType [ctor] .
  op None : -> PlaneTypeNone [ctor] .

  class Plane | type : PlaneType, landingTime : TimeInf,
                landAt : DefOid, earliest : Time, target : Time,
                latest : Time, early : Cost, late : Cost,
                latePenalty : Cost, clock : Time, rate : Cost .

  class Runway | landed : OidTimeList, type : PlaneTypeNone,
                 prevStart : Time .

  var P RW : Oid .
  var OTL : OidTimeList .
  vars C C1 ER : Cost .
  var TI : TimeInf .
  var R T L E Lt Ta : Time .
  var PT : PlaneType .
  var PTN : PlaneTypeNone .
  var SSt : SystemState .
```

The function sep defines the separation times as seen below. The first argument to the function is the type of the previously assigned plane and the second argument is the type of the plane that is being assigned, i.e., separation between a small and a big plane is 3 time units, while separation between a big and a small plane is 1 etc.

```
  op sep : PlaneType PlaneTypeNone -> Nat .

  eq sep(PT, None)       = 0 .
  eq sep(small, big)     = 3 .
  eq sep(medium, big)    = 2 .
  eq sep(big, big)       = 1 .
  eq sep(small, medium)  = 2 .
  eq sep(medium, medium) = 1 .
  eq sep(big, medium)    = 1 .
  eq sep(PT, small)      = 1 .
```

The rule `assignToNextFree` tries to assign a landing time to a plane based on the next free time slot, this may be earlier or later than the plane's target time. Because planes may fly around freely and unassigned up until their latest possible landing time.

```
crl [assignToNextFree] :
     < P : Plane | type : PT, earliest : E, landAt : noOid, latest : Lt,
                   clock : R, late : LR, early : ER, target : Ta >
     < RW : Runway | type : PTN, prevStart : T >
     =>
     < P : Plane | landAt : RW, landingTime : L, rate : C >
     < RW : Runway | type : PT, prevStart : L >
```

If a plane is assigned to the current time and this time is early some cost must be added to the system due to acceleration to land on time.

```
      with cost (if L == R and L =/= E then
                totEarlyCost(Ta, L, E, ER)
              else
                0
              fi)
  if L :=
          if R >= (T + sep(PT, PTN)) then
              R
          else
              T + sep(PT, PTN)
          fi
```

It may be necessary to adjust the rate of a plane at assignment time. If a plane is assigned earlier than target and the current time is between target and earliest the rate of the plane must be adjusted to take this into account.

```
     /\
     C := (if R < Ta and R >= E and L < Ta then
           --- accelerated rate equals total cost from earliness
           --- / the time interval left to accelerate
           (totEarlyCost(Ta, L, E, ER) / (Ta monus R))
          else
           0
          fi)
    /\ L >= E  /\ L <= Lt .
```

The rule `assignToTarget` tries to match a runway to an unassigned plane so the latter may be able to land on its target time. This may be achieved if the previous landing time assigned on the runway plus the necessary separation time is earlier or equal to the said plane's target time. In the case separation time plus previous landing time is earlier than the target landing time of a plane, this rule is used to "pad" the new previous landing so that it matches this plane's target time, where `assignToNextFree` would add the separation time to the current previous landing and might not hit a plane's target time, this rule will always match the plane's target time.

From this we see that the two rules `assignToTarget` and `assignToNextFree` , in conjunction with the fact that a plane may continue flying unassigned up until its latest landing time, makes this method complete in covering all possible landing scenarios.

```
  crl [assignToTarget] :
      < P : Plane | type : PT, latePenalty : C, earliest : E,
                    landAt : noOid, latest : Lt, target : Ta,
                    clock : R >
      < RW : Runway | type : PTN, prevStart : T >
      =>
      < P : Plane | landAt : RW, landingTime : Ta >
      < RW : Runway | type : PT, prevStart : Ta >
 if Ta >= (T + sep(PT, PTN)) .
```

Landing is completed by the rule `planeLanding` when a plane's clock reaches its assigned landing time. At this point the plane taxies down (at no further time-lapse) at the correct runway indicated by the plane's `landAt` attribute. During landing procedures, whether a plane lands on its earliest possible landing time, or if the landing time is past its target time will be determined. In the first case, the full cost of accelerating is incurred - i.e. the whole value of `early`. While in the second case, the one-time late penalty is incurred. When a plane has satisfactorily completed all landing sequences, its name and time of landing is added to the runway's landing log, and the plane object is discarded as it is no longer needed.

```
 rl [planeLanding] :
    < P : Plane | landAt : RW, landingTime : L, clock : L,
                  earliest : E, target : Ta, early : C, latePenalty : C1 >
    < RW : Runway | landed : OTL >
    =>
    < RW : Runway | landed : (OTL (P, L)) >
    with cost
    (if Ta < L then C1 else free fi
     pluss
 if L == E and Ta =/= E then C else free fi) .
```

Time flow is handled by the tick rule `tick` and the functions `delta`, `mte`, and `rate`. The tick rule uses `mte` to figure out how much time may elapse in the system. The function `rate` defines the current rate of cost per time unit so that it can be multiplied by the number of units advanced. Finally, `delta` is used in exactly the same way as in Real-Time Maude [23] to model the flow of time on the system.

```
  crl [tick] :
      {SSt} => {delta(SSt, R)} in time R with cost (rate(SSt) * R)
  if R <= mte(SSt) [nonexec] .

  --- Runways do not generate cost or affect time
  eq delta(< RW : Runway | >, R:Time)
     =
     < RW : Runway | > .
  eq mte(< RW : Runway | >) = INF .
  eq rate(< RW : Runway | >) = 0 .
```

The function `mte` is set up so the system cannot advance past a point in time where the rate of a plane may change. The rate may change at the earliest and the target landing time depending on the assigned landing time.

Unassigned planes are assumed to land on target until time passes this time. Planes may stay unassigned up until their latest landing time. However. time must be stopped at a plane's target time to change to the late rate.

```
eq mte(< P : Plane | landAt : noOid, earliest : E, target : Ta,
                     clock : R, latest : L >) = if R < Ta then
                                                 Ta monus R
                                                else
                                                 L monus R
                                                fi .


--- any assigned plane before earliest
ceq mte(< P : Plane | landAt : RW, clock : R, landingTime : L, earliest : E >)
    = E monus R
if R < E .

--- landing time
eq mte(< P : Plane | landAt : RW, clock : L, landingTime : L >) = 0 .

--- landing on target
eq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : T >) = T monus R .

--- early before target
ceq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : L, earliest : E >)
    =  L monus R
if E < T /\ R < T  /\ R >= E .

--- late between earliest and target
ceq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : L, earliest : E >)
    = T monus R
if L > T /\ R >= E /\ R < T .

--- late between target and landing time
ceq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : L >)
     = L monus T
if L > T /\ R >= T .
```

The function `delta` updates the clock on each plane.

```
eq delta(< P : Plane | clock : L >, R)
   =
   < P : Plane | clock : L plus R > .
```

The `rate` function calculates the rate of a plane based on its assigned landing time and the current time. The rate of all planes is 0 until their earliest possible landing time is reached. The plane uses the late rate when its time goes beyond the target time. Early planes have a rate calculated proportionally to how close their assigned landing time is to the earliest landing time. Planes that are assigned to land on target will never incur any cost, therefore, their rate is always 0.

```
eq delta(< P : Plane | clock : L >, R)
   =
   < P : Plane | clock : L plus R > .

--- If time is before earliest there's no rate
ceq rate(< P : Plane | earliest : E, clock : R >) = 0
if R < E .
```

```
  --- If plane is scheduled to be on target there's no rate
  ceq rate(< P : Plane | clock : R, landingTime : L, target : L >) =  0 if R < L .

  --- if target time has been reached the rate is late
  ceq rate(< P : Plane | late : C, target : Ta, clock : R >) = C
  if R >= Ta .
```

The closer a plane is assigned to its earliest landing time the higher the rate in the interval between earliest and target. If a rate has already been set at the time of assignment a new rate will not be calculated.

```
  ceq rate(< P : Plane | landAt : RW, earliest : E, early : ER,
                         target : Ta, landingTime : L, clock : R, rate : C >)
      =  if C == 0 then
         --- cost per time unit equals the total early cost divided by the
         --- interval between target and landing
         (totEarlyCost(Ta, L, E, ER) / (Ta monus L))
         else
          C
         fi
  if R >= E and L < Ta and R =/= Ta .
```

The function `totEarlyCost` computes the total cost a plane will incur in the system due to an early landing.

```
 op totEarlyCost : Time Time Time Cost -> Cost .
 eq totEarlyCost(Ta, L, E, ER) = ((Ta monus L) / (Ta monus E)) * ER .
endptom)
```

7.1.4 ALP Analysis in Priced-Timed Maude

We now want to use our ALP specification to determine optimal landing schedules. We can use Priced-Timed Maude to find a state with the cheapest possible cost and read the schedule out of the runway.

Defining an Initial State

The following Priced-Timed Maude module defines the 3 planes and the runway of in Example 7.1.1:

```
(ptomod ALP-TEST is
 protecting ALP .
 protecting STRING .

 subsort String < Oid .

 op plane : Oid PlaneType Time Time Time Cost Cost Cost -> Configuration .
 op runway : Oid -> Configuration .
 op init1 : -> GlobalSystem .

 vars E T L : Time .
 vars C1 C2 C3 : Cost .
 var PT : PlaneType .
 var O : Oid .
```

```
eq plane(O, PT, E, T, L, C1, C2, C3) =
         < O : Plane | type : PT, landAt : noOid, landingTime : T,
                       clock : 0, early : C1, late  : C2, latePenalty : C3,
                       earliest : E, target : T, latest : L, rate : 0 > .
```

The function `plane` is used to eliminate repeating values that are the same for all planes, as well as, simplifying the task of instantiating a new plane. We can instantiate a new plane by using the `plane` function giving: a name, type, earliest landing time, target landing time, latest landing time, early rate, late rate, and a late penalty.

```
eq runway(O) =  < O : Runway | landed : nil, type : None, prevStart : 0 > .

--- 3 planes and one runway
eq init1 = {plane("p1", small, 5, 9, 12, 15, 2,3)
            plane("p2", small, 5, 5, 10, 25, 5, 10)
            plane("p3", medium, 5, 7, 15, 20, 3, 5)
            runway("rw1")} .
endptom)
```

Determining Earliest and Cheapest Schedule using Priced-Timed Maude

By using the Priced-Timed Maude command, `priced find earliest`, we confirm that S_1 of Example 7.1.2 is the earliest schedule:

```
Maude> (priced find earliest init1 =>* {< "rw1" : Runway | >} with no cost limit.)

Result: {< "rw1" : Runway | landed :(("p1",5)("p2",6)("p3",7)),prevStart : 7,
                            type : big >} in time 7 with cost 30
```

Likewise, by using the command `find cheapest` we confirm that S_2 is the optimal schedule with regard to cost:

```
Maude> (find cheapest init1 =>* {< "rw1" : Runway | >} with no time limit .)

Solution
ATTRIBUTES_OF_"rw1":AttributeSet -->
 landed :(("p2",5)("p3",7)("p1",9)),
 prevStart : 9,type : small ;
CLASS_OF_"rw1":Runway --> Runway ;
TIME_ELAPSED:Time --> 9 ; TOTAL_COST_INCURRED:Cost --> 0
```

Clearly, S_2 is an optimal schedule with regard to cost, while S_1 is optimal with regard to time.

Defining more Planes and Runways

We can easily define initial states with more planes and runways by adding another constant of sort `GlobaleSystem` to `ALP-TEST` by doing the following:

```
ops init1 init2 : -> GlobalSystem .
```

We can also define a new initial state `init2` with 6 planes and 3 runways like the following:

```
eq init2 = plane("p1", small, 3, 4, 7, 15, 2,3)
           plane("p2", medium, 0, 2, 10, 20, 3, 5)
           plane("p3", big, 2, 4, 5, 25, 5, 10)
           plane("p4", medium, 2, 4, 7, 15, 3, 7)
           plane("p5", medium, 1, 2, 6, 15, 3, 5)
           plane("p6", big, 2, 5, 7, 25, 7, 10)
           runway("rw3")
           runway("rw2")
           runway("rw1") .
```

We are now free to use all the same tools provided by Priced-Time Maude as we were on the previous initial state `init1`. For instance, we may now use the optimal priced timed search command `find cheapest` to find an optimal schedule which in this case should be reachable in time 4 with cost 0.

7.2 Energy Task Graph Scheduling

The second problem is *energy task graph scheduling* (ETGS) [13]. In this problem we are given a set of interdependant tasks, a set of processors, and a bus. The tasks require a given time to run on each processor and to broadcast on the bus. The processors and bus consume power at a given rater while processing/broadcasting, and a different rate of power while idle. The objective is to find the most energy efficient schedule for completing all the tasks within a given deadline.

This example was chosen because of its practical applications and it is the one cited in [13] along with specification techniques for UPPAAL *CORA* [10] an automata based priced-timed tool. This gives us the opportunity for comparison between the specification languages and performance of Priced-Timed Maude and UPPAAL *CORA* [10] in Chapter 8.

7.2.1 Energy Task Graphs and Examples

Task graphs are used to solve problems where there are constraints on the ordering of a set of tasks and resources may be limited.

Example 7.2.1 *Consider the project of building a house. Some of the tasks may include building a foundation, building walls, putting in doors and windows, and installing pipes. The resources may include specialist manpower such as plumbers, carpenters, and masons. All tasks would depend on the foundation being laid down first, while putting up the walls and laying pipes may be done in parallel provided there is enough personnel.*

A more specialized version of the task graph is the energy task graph that arises when dealing with scheduling problems on embedded systems. On some embedded systems. battery life is often short, and speed might not always be the most important factor to solving a problem, but running the system for as long as possible within certain performance parameters. Take for instance a portable media device such as an mp3 player: It is not important to play the mp3s as quickly as possible, merely to play at a given quality and avoid any stuttering, i.e., play at a given preset speed while ensuring the longest possible battery life.

An energy task graph is used to describe a system with an ordering of interdependant tasks, a bus and a set of processors. In addition, the tasks have given running times on the different processors and time required to broadcast on the bus. The energy part of the energy task graph is the idle and processing power consumption of the different processors and the bus.

Definition 7.2.2 (Energy task graph) *(from [13]) An energy task graph is a tuple* $(T, P, pre, \delta, \kappa, \pi, \tau, d)$ *and a* bus *where*

- $T = \{t_1, ..., t_n\}$ *is a set of* tasks.
- $P = \{p_1, ..., p_m\}$ *is a set of* processors. *In addition, there's an element* bus *that can be considered a special processor that is used to broadcast the result of tasks to all processors.*
- $pre : T \to 2^T$ *determines the set of* predecessors *of each task.*
- $\delta : T \times P \to \mathbb{N}$ *is the execution time for tasks on processors.*
- $\kappa : T \to \mathbb{N}$ *is the transfer for each task; that is the time required to broadcast the result of a task on the bus.*
- $\pi : P \cup \{bus\} \to \mathbb{N}$ *is the energy consumption rate per time unit for active processors/bus.*
- $\tau : P \cup \{bus\} \to \mathbb{N}$ *is the energy consumption for processors/bus when idle.*
- d *is the deadline.*

We will use the following shorthand notation for some of the above terms:

- pre_i for $pre(t_i)$.
- $\delta_{i,j}$ for $\delta(t_i, p_j)$.
- κ_i for $\kappa(t_i)$.
- $\{\pi, \tau\}_i$ or π_i, τ_i for $\pi(t_i)$ and $\tau(t_i)$.
- $\{\pi, \tau\}_{bus}$ or π_{bus}, τ_{bus} for $\pi(bus)$, and $\tau(bus)$.
- Let $\{\pi, \tau\}$ denote the set $\{\{\pi, \tau\}_1, ..., \{\pi, \tau\}_k, \{\pi, \tau\}_{bus}\}$.

Example 7.2.3 *An energy task graph $\mathcal{E}$ with 3 tasks, 2 processors and a bus $\mathcal{E} = (T, P, pre, \delta, \kappa, \pi, \tau, d)$ can be given as follows:*

1. $T = \{t_1, t_2, t_3\}$
2. $P = \{p_1, p_2\}$,
3. $pre_1 = \{\emptyset\}$
4. $pre_2 = \{\emptyset\}$
5. $pre_3 = \{t_1, t_2\}$
6. $\delta = \{\delta_{1,1} = 1, \delta_{3,1} = 5, \delta_{2,2} = 2, \delta_{3,2} = 4\}$
7. $\kappa = \{\kappa_1 = 7, \kappa_2 = 5\}$
8. $\{\pi, \tau\} = \{\{5, 1\}_1, \{4, 1\}_2, \{11, 1\}_{bus}\}$
9. $d = 12$

We see that t_3 depends on the results of t_1 and t_2. t_1 can run only on p_1 and uses 1 millisecond (ms) on p_1, while broadcasting the result on the bus takes 7 ms. t_2 may only be run on p_2 and uses 2 ms, broadcasting the result of t_2 on the bus takes 5 ms. t_3 may run on either processor, requires 5 ms to process on p_1 or 4 on p_2. The bus consumes 11 units of power per time unit while broadcasting and 1 while idle. p_1 uses 5 units of power while active and 1 while idle. p_2 consumes 4 units of power while active and 1 while idle.

Feasible Schedules and Notation

To solve the problem of finding an optimal schedule we first need to know what schedules are feasible. The reference [13] gives a definition of this; the following is an informal recap of the main points:

(1) Tasks can only execute on allowed processors and the result cannot be broadcast until execution is terminated.

(2a) When a task depends on the result of another task, these are either executed on the same machine or the result of the earlier task has to be broadcast.

(2b) No task can begin executing until the results of all dependant tasks are available.

(2cd) Each processor/bus can only execute/transfer one task/result at any given time and such operations cannot be preempted.

(3) The schedule S must meet the deadline d, all tasks must be completed at time less than or equal to d.

Example 7.2.4 *A feasible schedule S_1 for the ETGS $\mathcal{E}$ is that t_1 started processing on p_1 at time 0, t_2 on p_2 at time 0 then broadcast the result of t_1 at time 1, finally t_3 started processing on p_2 at time 8. From this we see that S_1 meets the deadline with t_3 starting at 8 and requiring 5 units of time to process it will be done at time 12. A second schedule S_2 starts t_1 and t_2 on p_1 and p_2 in the same manner, but instead of broadcasting the result of t_1 we wait for t_2 to finish running on p_2, the result of t_2 is broadcast at time 2. S_2 then runs t_3 on p_1 at time 7 and finishes t_3 at time 12.*

It is easy to see that S_1 and S_2 are both feasible schedules as they finish all tasks within the given deadline. However, only one of them is optimal with regard to power consumption. This will be discussed in greater detail in the next section.

Cost of Schedules and Optimal Schedules

We now know what a *feasible* schedule looks like, but to determine an *optimal* schedule we first need to know how to determine the cost of a schedule. Intuitively, this is easy: The cost of a schedule S must certainly be the cost accumulated through the schedule by all components while processing and idling.

First, we need to define the processing time for the processors and bus:

Definition 7.2.5 *The processing time of a given processor p_k denoted $proc(p_k)$ is the sum of the execution times of the tasks that ran on it during a schedule; likewise, the processing time of the bus $proc(bus)$ is the sum of the broadcast times of tasks that were broadcast.*

Example 7.2.6 *For the schedule S_1 $proc(p_1) = \delta_{1,1} = 1$, $proc(p_2) = \delta_{2,2} + \delta_{3,1} = 6$ and $proc(bus) = \kappa_1 = 7$ For S_2 $proc(p_1) = \delta_{1,1} + \delta_{3,1} = 6$, $proc(p_2) = \delta_{2,2} = 2$ and $proc(bus) = \kappa_2 = 5$*

It is easy to see that the idle time of a processor or bus is the running time of the schedule minus the processing time. For simplicity, let's assume that the running time of a schedule is the deadline d; therefore, $idle(p_k) = d - proc(p_k)$.

Example 7.2.7 *Now we want to figure out the idle times of the different components of S_1 and S_2: For S_1: $idle(p_1) = 12 - 1 = 11$, $idle(p_2) = 12 - 6 = 6$ and $idle(bus) = 12 - 7 = 5$. For S_2: $idle(p_1) = 12 - 6 = 6$, $idle(p_2) = 12 - 2 = 10$ and $idle(bus) = 12 - 5 = 7$.*

It is now easy to define the cost of each component during a schedule:

Definition 7.2.8 *The cost accumulated by a component during a schedule is the amount of power it consumes during the schedule while idle plus the power consumed while active i.e. $cost(p_i) = \pi_i \cdot proc(p_i) + \tau_i \cdot idle(p_i)$ in the same way $cost(bus) = \pi_{bus} \cdot proc(bus) + \tau_{bus} \cdot idle(bus)$*

Example 7.2.9 *Computing the cost for each component of S_1 and S_2: The cost of S_1's components: $cost(p_1) = \pi_1 \cdot proc(p_1) + \tau_1 \cdot idle(p_1) = 5 \cdot 1 + 1 \cdot 11 = 16$, $cost(p_2) = 24 + 6 = 30$ and $cost(bus) = 77 + 5 = 82$. For S_2 $cost(p_1) = 30 + 6 = 36$, $cost(p_2) = 8 + 10 = 18$ and $cost(bus) = 55 + 7 = 62$.*

Now we are ready to define the cost of a schedule

Definition 7.2.10 (Cost of a schedule) *The cost of a schedule S is the sum of the cost of all its components:*

- $Cost(S) = \sum_{p_k \in P} (\pi_k \cdot proc(p_k) + \tau_k \cdot idle(p_k)) + \pi_{bus} \cdot proc(bus) + \tau_{bus} \cdot idle(bus)$

Example 7.2.11 *The cost of S_1 and S_2:*

- $Cost(S_1) = 16 + 30 + 82 = 128$
- $Cost(S_2) = 36 + 18 + 62 = 116$

In addition to what has already been defined, we also want to keep track of the rate, the power consumption per time unit at any given time in the system. The rate is the sum of power consumption of all units at any given moment. This information helps us get an overview of the current power consumption of the system at any given moment in time.

Example 7.2.12 *The following two diagrams show both schedules, each broken down by activities and rate: first S_1:*

time	p1	p2	bus	*rate*	*duration*	*cost*
0	t1	t2	*idle*	*10*	*1*	*10*
1	*idle*	t2	t1	*16*	*1*	*16*
2	*idle*	*idle*	t1	*13*	*6*	*78*
8	*idle*	t3	*idle*	*6*	*4*	*32*
12						*128*

and then S_2:

time	p1	p2	bus	*rate*	*duration*	*cost*
0	t1	t2	*idle*	*10*	*1*	*10*
1	*idle*	t2	*idle*	*6*	*1*	*6*
2	*idle*	*idle*	t2	*13*	*5*	*65*
7	t3	*idle*	*idle*	*7*	*5*	*35*
12						*116*

The column headings in the tables denote the current activity of p1*,* p2*,* p3*, and the bus, respectively; as well as, the time an action starts, rate and duration thereof; its concurrent action; and the total cost during that period. For instance, row 3 of the first table states that at time 2,* p1 *and* p2 *are idle while the bus is broadcasting the result of* t1*. This has a rate of 13 mW/ms and lasts for 6 ms for a total of 78 mW consumed. The total duration and cost of the schedule is given in the last row and is 12 ms and 128 mW for S_1 and 12 ms and 116 mW for S_2.*

Finally, we define optimal schedules in the following way:

Definition 7.2.13 (Optimal schedule for the ETGS problem) *An optimal schedule in the ETGS problem is a schedule S^* such that $cost(S^*) \leq cost(s)$ for all schedules $S^*, s \in S$ such that the set S includes only those feasible schedules where all tasks are finished.*

Using this definition we can easily determine that S_2 is an optimal schedule, as any other order of actions than S_1 or S_2 would contain redundant actions such as broadcasting t_1 or t_2 while t_3 is being processed.

7.2.2 Modeling ETGS In Priced-Timed Maude

We can specify an ETGS problem in Priced-Timed Maude by defining a class for tasks: one class for processors and one for buses. Since buses and processors will share most attributes, these can both be subclasses of a common superclass. The class `ProcDevice`, which is the superclass for processors and buses, needs the following attributes:

- `currentTask`: which task is currently being processed/broadcast.

- `activeRate`: the rate at which power is consumed when the device is busy.
- `idleRate`: the rate at which a processor/bus consumes power while idle.
- `timer`: keeps track of the remaining running/broadcast time of the current task. This has value `INF` while the device is idle.

The class `Processor`, which is a normal processor, will need just one extra attribute `knownRes`. This should be a set of `Oid`s for the tasks that the processor has the results for.

The class `Bus` that models a bus needs one additional attribute `connectedTo`, a set with the `Oid`s of the processors a bus is connected to.

The tasks can be modeled by a class `Task` with these attributes:

- `dependsOn`: a set containing `Oid`s of all the tasks a task depends on;
- `procTime`: a set of pairs of the form `(Oid, Time)` where the `Oid` refers to a processor and the `Time` is the required time for executing on that processor;
- `bcastTime`: the amount of time required to broadcast the result of the task on a bus;
- `status`: this should be a status with one of these values: `unprocessed`, `processing`, or `done`.

Furthermore, the rules of the system must satisfy the requirements of the definition of feasible schedules given in Section 7.2.1 (Feasible Schedules and Notation). Note that a deadline is not explicitly given in the Priced-Timed Maude specification as this will be dealt with when executing the specification.

The following Priced-Timed Maude specification models the ETGS problem:

The imported module `OID-SET` provides us with a sort `OidSet`, which is a set of `Oid`s. In addition to normal set operators, the module also defines the `subset` operator that calculates whether a set is a subset of another and returns true or false based on this.

```
(omod OID-SET is
  protecting DEF-OID .
  protecting SET{Oid} * (sort Set{Oid} to OidSet,
                         sort NeSet{Oid} to NeOidSet) .

  vars OS OS' : OidSet .

  op _subset_ : OidSet OidSet -> Bool .
  eq OS subset (OS, OS') = true .
  eq OS subset OS' = false [owise] .
endom)
```

The imported module `OID-SET` defines the basic message type we need for broadcasting a result on the bus. The first `Oid` represents the task that the result is being broadcast for, while the second represents the receiver (a processor).

```
(omod MESSAGES is
  sort MsgType .

  op bcast : -> MsgType [ctor] .

  msg msg_from_to_ : MsgType Oid Oid -> Msg .
endom)
```

`MESSAGES-MULT1` provides us with the means to dispatch broadcast messages, e.g., sending the result of a task on a bus to all the processors that are connected to the bus.

```
(omod MESSAGES-MULT1 is
  protecting MESSAGES .
  protecting OID-SET .

  op multimsg_from_to_ : MsgType Oid OidSet -> Configuration .

  var MT : MsgType .
  vars O O' : Oid .
  var OS : OidSet .

  --- Make a message for every Oid in the set
  eq multimsg MT from O to empty = none .
  eq multimsg MT from O to (O', OS) =
     (msg MT from O to O') (multimsg MT from O to OS) .

endom)

(tomod OID-TIME is
  sort OidTimePair .
  op _`,_ : Oid Time -> OidTimePair [ctor] .
endtom)

(view OidTime from TRIV to OID-TIME is
  sort Elt to OidTimePair .
endv)
```

Lastly, the module `OID-TIME-SET` provides us with sets that consist of pairs of the form `Oid, Time`), i.e., the previously described sort needed for specifying the running time of a task on a processor.

```
(tomod OID-TIME-SET is
  protecting SET{OidTime} * (sort Set{OidTime} to OidTimeSet,
                             sort NeSet{OidTime} to NeOidTimeSet) .
endtom)

(ptomod ETGS is
  protecting OID-TIME-SET .
  protecting MESSAGES-MULT1 .
  protecting NAT-COST-DOMAIN .
  protecting NAT-TIME-DOMAIN-WITH-INF .
```

The modules `NAT-COST-DOMAIN` and `NAT-TIME-DOMAIN-WITH-INF` set cost and time to run over the natural numbers.

The sort `TaskState` is used to indicate the state of a task as unprocessed, processing, or done.

```
  sort TaskState .

  ops unprocessed processing done : -> TaskState [ctor] .
  class ProcDevice | currentTask : DefOid,
                     activeRate  : Cost,
                     idleRate    : Cost,
                     timer : TimeInf .

  class Task | dependsOn : OidSet,
               procTimes : OidTimeSet,
```

```
              bcastTime : Time,
              status    : TaskState .

  class Processor | knownRes : OidSet .
  subclass Processor < ProcDevice .
```

A `Bus` is a special processing device that connects processors. It can be used in a similar way to a processor to broadcast the result of a task finished on one its connected processors to all of its connected processors.

```
  class Bus | connectedTo : OidSet .
  subclass Bus < ProcDevice .

  var R : Time .
  var TI : TimeInf .
  var SSt : SystemState .
  vars T P B : Oid .
  var C : Cost .
  var OS OS' : OidSet .
```

The rule `startTaskOnProcessor`: this rule will start an unprocessed task on a free processor for as long as it has an entry specifying its running time on this processor. The task's status is set to `processing` and the processor is set busy by indicating the task's `Oid` in the `currentTask` attribute. In addition, the timer of the processor is set to the running time of the task. This prevents any other tasks from running on the same processor as well as preventing this task from running on any other processors as a task needs to have the `unprocessed` status to be initiated. Furthermore, a task cannot start running on a processor unless the results it depends on exist in the `knownRes` set of the processor.

```
  crl [startTaskOnProcessor] :
      < T:Oid : Task | status : unprocessed,
                       procTimes : ((P:Oid, R:Time),
                                    OTS:OidTimeSet),
                       dependsOn : OS:OidSet >
      < P : Processor | timer : INF, knownRes : OS':OidSet >
      =>
      < T : Task | status : processing >
      < P : Processor | currentTask : T, timer : R >
  if (OS:OidSet subset OS':OidSet) .
```

The rule `finishTaskOnProcessor` makes sure that when a task is done running on a processor, the result is known by that processor. After which, the processor is set back to idle and the task's status is `done`.

```
  rl [finishTaskOnProcessor] :
     < T : Task | status : processing >
     < P : Processor | currentTask : T, timer : 0, knownRes : OS >
     =>
     < T : Task | status : done >
     < P : Processor | currentTask : noOid, knownRes : (OS:OidSet, T),
                       timer : INF > .
```

The rule `startBroadcast` starts a broadcast on a bus with a task, this requires the task to have the status `done` and the bus to be free. When a broadcast is started, the bus' timer is set using the `bcastTime` of the task and the `Oid` of the task as the currently broadcasting task on the bus. Tasks that have been broadcast are discarded as they are no longer needed for anything.

```
rl [startBroadcast] :
   < T : Task | status : done, bcastTime : R >
   < B : Bus | timer : INF >
   =>
   < B : Bus | currentTask : T,
               timer : R > .
```

`finishBroadcast` sends out a message to all connected processors once a bus' timer is 0, then sets the bus back to idle status.

```
rl [finishBroadcast] :
   < B : Bus | currentTask : T, timer : 0,
               connectedTo : OS >
   =>
   < B : Bus | currentTask : noOid, timer : INF >
   multimsg bcast from T to OS .
```

The `receiveBroadcast` rule, on the other hand, deals with a processor receiving a broadcast. When a broadcast message with a result is received, the receiving processor updates its `knownRes` with the new result.

```
rl [receiveBroadcast] :
   (msg bcast from T to P)
   < P : Processor | knownRes : OS >
   =>
   < P : Processor | knownRes : (OS, T) > .
```

Lastly, we have a fairly standard tick rule, `[tick]`, that models the elapse of time on the system.

```
crl [tick] :
    {SSt} => {delta(SSt, R)} in time R with cost (rate(SSt) * R)
if R <= mte(SSt) [nonexec] .
```

The function `mte` determines the maximum amount of time that can elapse. While a message is in the system no time may elapse; otherwise, it is dependant on the time left on running tasks on any processor or bus.

```
eq mte(M:Msg) = 0 .
eq mte(< T : Task | >) = INF .
eq mte(< P : ProcDevice | timer : TI >) = TI .
```

The function `delta` works in the same manner as in Real-Time Maude [2], updating all the timers according to the time elapsed. The `rate` function extracts an energy consumption rate from all the components in the system and adds them up to one global rate that can be multiplied by the time elapsed.

```
eq delta(< T : Task | >, R)
         =
         < T : Task | > .
--- Processors and buses are affected by time.
--- A processor or bus' timer counts down during the elapse of time.
eq delta(< P : ProcDevice | timer : TI >, R)
   =
   < P : ProcDevice | timer : (TI monus R) > .
```

Only processors and buses have a rate, the `rate` function extracts the idle or active rate from these according to what state they are currently in. If a bus/processor's `timer` equals `INF` the idle rate is used, otherwise the active rate is used.

```
  eq rate(< P : ProcDevice | idleRate : C, timer : INF >) = C .
  eq rate(< P : ProcDevice | activeRate : C, timer : R >) = C .
  eq rate(< T : Task | >) = 0 .
endptom)
```

7.2.3 ETGS Execution And Analysis

In this section the ETGS problem given in Example 7.2.3 is analyzed using Priced-Timed Maude.

The following module defines an initial state with 3 tasks, 2 processors, and a bus as described in Example 7.2.3:

```
(ptomod ETGS-TEST is
 protecting NAT .
 protecting ETGS .
 protecting STRING .

 subsort String < Oid .

 op init : -> GlobalSystem .

 eq init = {< "t1"   : Task | dependsOn : empty, procTimes : ("p1", 1),
                              bcastTime : 7, status : unprocessed >
            < "t2"   : Task | dependsOn : empty, procTimes : ("p2", 2),
                              bcastTime : 5, status : unprocessed >
            < "t3"   : Task | dependsOn : ("t1", "t2"),
                              procTimes : (("p1", 5),  ("p2", 4)),
                              bcastTime : 0, status : unprocessed >
            < "p1" : Processor | currentTask : noOid, timer : INF,
                                 knownRes : empty, activeRate : 5,
                                 idleRate : 1 >
            < "p2" : Processor | currentTask : noOid, timer : INF,
                                 knownRes : empty, activeRate : 4,
                                 idleRate : 1 >
            < "bus"    : Bus | currentTask : noOid, timer : INF,
                               connectedTo : ("p1", "p2"),
                               activeRate : 11, idleRate    : 1 >} .
endptom)
```

For this system we wish to find an *optimal schedule* starting from the initial state `init` to a state where all tasks are finished within 12 ms.

Using Priced-Timed Maude to find An Optimal Schedule

We now use the Priced-Timed Maude command `find cheapest` to verify that the second schedule is in fact the optimal one for a 12 ms deadline by executing the following:

```
Maude> (find cheapest init =>* {S:SystemState < "t3" : Task | status : done >} in time <= 12 .)
```

```
Solution
CLASS_OF_"t3":Task --> Task ;
    REMAINING_ATTRIBUTES_OF_"t3":AttributeSet -->
        bcastTime : 0, dependsOn :("t1", "t2"), procTimes :("p1",5, "p2",4);
S:SystemState -->
    < "bus" : Bus | activeRate : 11, connectedTo :("p1","p2"), currentTask : noOid,
                    idleRate : 1, timer : INF >
    < "p1" : Processor | activeRate : 5,currentTask : noOid, idleRate : 1,
                         knownRes :("t1", "t2", "t3"), timer : INF >
    < "p2" : Processor | activeRate : 4, currentTask : noOid, idleRate : 1,
                         knownRes : "t2", timer : INF >
    < "t1" : Task | bcastTime : 7, dependsOn : empty, procTimes : "p1",1, status : done > ;
    TIME_ELAPSED:Time --> 12 ; TOTAL_COST_INCURRED:Cost --> 116
```

This confirms that the schedule S_2 is an optimal schedule. Note that we only need to search for any state where `t3` is done since it relies on the fact that the other tasks are already done. It is also notable that we would have arrived at the same result if we searched for this term with no time limit, since this is the optimal result for a state where all tasks are finished.

7.3 Subway Passenger Routing

Finally, the *subway passenger routing* (SPR) problem routing passengers who want to travel within a subway network. Each train uses a set amount of power based on how many cars are attached to it. The objective is to minimize the trains' total power consumption, while at the same time making sure all passengers reach their destinations.

This problem is presented mainly to illustrate how more complex datatypes than traditional automata based tools for analyzing priced-timed systems can be used in Priced-Timed Maude. Also, it is to illustrate how complex conditions on rules and equations can be when using Priced-Timed Maude, enabling us to specify any number of arbitrary functions as conditions.

7.3.1 Passenger Routing in A Subway System

This problem is inspired by and loosely based on the challenge of routing railway stock in which the minimum amount of cars needed to service a given schedule has to be determined based on the specific expectation of passenger load on a German subway network. This problem, is Application 4.5 in [24].

For this particular situation, the following are given: a subway map, subway trains assigned to specific routes within that map, and schedules that determine the flux of passengers for each station during a given day. The map provides a grid display of the connectivity of the different stations and the relative travel times between them.

The route assigned to each subway train represents its (directed) path along the map. When a train reaches the end of its route, it turns around and travels the same way, albeit in reverse. Furthermore, each train consumes a certain amount of power - this figure is based on the number of cars attached to the train.

Each station has a predetermined schedule of passenger arrivals from outside the system. This determination comes as a result of regular surveys conducted to ascertain traffic patterns. (The passengers that arrive at each station are assumed to know their next destinations.) In addition, at certain stations called service stations, the trains will be allowed to detach cars to save power or attach more cars to make room for more passenger in order to save time.

Finally, commuters should be able to get to their destination by the most direct route possible. This means not boarding a train going in the wrong direction. If a passenger does this then he has to pass by the station where he boarded after the train turns at its end station.

The objective is to determine the most effective method of transporting all the commuters to their respective destinations in the most energy or time-efficient manner.

7.3.2 A Subway Network Example

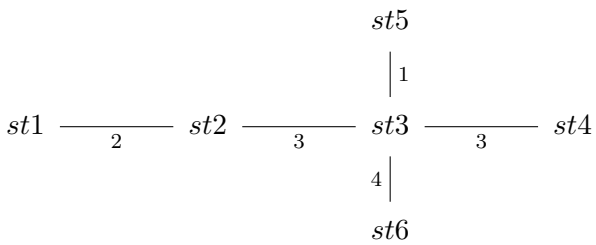


Figure 7.1: A map of a subway with 6 stations; the traveling times between the stations are denoted on the edges between them.

To clarify the point of a passenger not boarding a train in the wrong direction, consider Figure 7.1, which shows the layout and travel times in a subway network with 6 stations denoted `st1` to `st6`. A station A is reachable from a station B if there is a *directed* path connecting them. In the map in Figure 7.1 any station is reachable from any other station but, normally, a passenger would not board a train heading in the opposite direction of his desired destination. Consider the routes shown in Figure 7.2 : 1a, covering `st1` through `st4` and 1b, covering `st4` to `st1`.

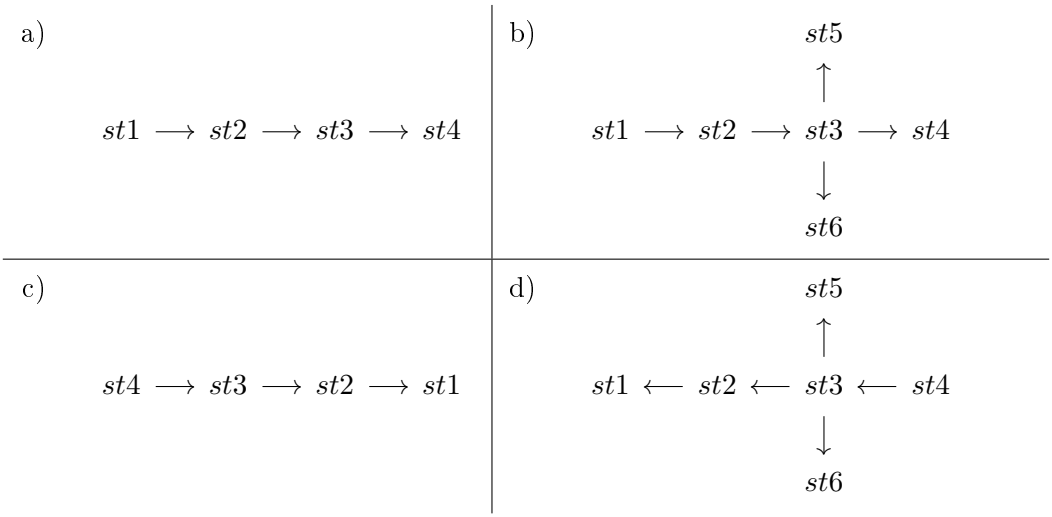


Figure 7.2: a) A route 1a goes from $st1$ to $st4$. b) The layout of the subway with regard to accessibility/transfers from route 1a. c) The route 1b simply traversing 1a backwards. d) The layout as seen from 1b.

Example 7.3.1 *Now let us consider a passenger with destination* `st4` *who is waiting for a train at the station* `st2`*. It is clearly counterproductive for this passenger to step onto a train traveling route 1a as this would not take him closer to his destination. Instead, he may wish to board a train servicing the route 1b that would take him to* `st4` *via* `st3`*.*

Example 7.3.1 illustrates that when determining the reachability of one station from another, we need to consider this problem from the perspective of a traveling train and take into account that train's route. Figure 7.2 d shows how the subway layout is seen as a directed graph relative to route 1b in terms of its reachability to stations. In determining reachability in a directed graph like this one, a station A is reachable from a station B only if there is a directed path from B to A. Put simply, if we can get from B to A by following the direction of the edges, A is reachable from B. From hereon, when the word path is mentioned in connection with reachability, it refers to a *directed path* like we just described.

Example 7.3.2 *By examining Figure 7.2 part d, we easily determine that there is no path from* `st2` *to* `st4` *when traveling on a train servicing route 1b. On the other hand, a train servicing route 1a contains a path from* `st2` *to* `st4`*. If, in this example, the passenger wanted to go from* `st2` *to* `st5` *or* `st6` *instead. he would have to get off at* `st3` *and wait for a train that services the routes 2a and 2b as shown in Figure 7.3 a and c.*

7.3.3 Modeling Passengers, Trains and Stations in Priced-Timed Maude

Passengers need to be modeled along with their destination. However, since it is not interesting nor practical to model passengers as separate and distinct objects with distinct identities, they will be

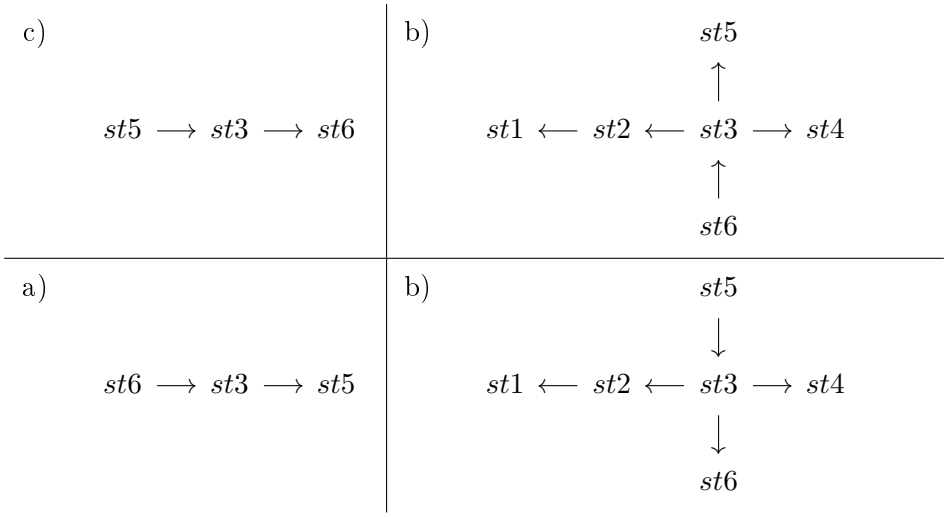


Figure 7.3: a) A route 2a goes from $st5$ to $st6$. b) The layout of the subway in terms of accessibility/transfers from route 2a. c) The route 2b simply traversing 2a backwards. d) The layout as seen from 2b.

modeled with a sort `PassengerGroup` which would represent a group of 1 or more passengers going to the same destination. Terms of the `PassengerGroup` sort has the form

Nz `going to` S

where Nz is a non-zero natural number and S is the name of a station. For instance, the term `1 going to "Anaheim"` is one passenger going to the station named Anaheim. Since stations and trains clearly need to accommodate several groups of passengers that may wish to travel to different locations, we need sets of `PassengerGroup` terms called a `PassengerSet`.

To model passengers that arrive at a station from outside the system at a certain time, we will need a new sort `PassengerSchedule` which represents a list of passengers coming in at a given time. The entries of this list be of a sort `PassengerScheduleEntry` that has the form

`(`PS`,` T`)`

where PS is a `PassengerSet` signifying the amount of boarding passengers and their destinations, while T is of sort `Time` and indicates when these passengers will be arriving. The terms of the `PassengerSchedule` sort have the form of a list of `PassengerScheduleEntry` entries giving the specific time sets of passenger arrivals. Note that this list should be sorted from the earliest arrivals to the latest ones.

We model stations by declaring a class `Station` with the following attributes:

- `trains`: a set of `Oid`s identifying the trains that are currently at the station.
- `passengers`: a set representing all the passengers currently at the station.
- `passengerSchedule`: a list that indicates when different passenger groups come into the station.
- `clock`: the current time.

Consequently, a class `SeStation` which is a subclass of `Station` is needed to model the service stations where cars may be attached and detached. For this class, an extra attribute `cars` that keeps track of the station's inventory of unattached cars is vital.

Trains will be modeled by a class `Train` with these attributes:

- `cars`: the number of subway cars this train consists of.
- `route`: a list of stations that make up a route for this train.
- `visited`: a list of stations visited by the train. (Generally, when a station in the route is visited, it is removed from the route and put here instead.)
- `status`: a status indicating the train's current activity, such as `traveling to station` A or `arrived at station` A.
- `timer`: time left of the train's current task, for instance as the train trudges on its route from A to B, and the timer indicates a 0, this means the said train has actually arrived at station B.
- `passengers`: a set of passengers currently in the train.
- `activeLO`: the subway map as a directed graph as viewed from the train's current orientation.
- `passiveLO`: a subway map as seen from the train when it switches directions.

The following Priced-Timed Maude specification implements the classes and sorts and model the problem as described:

The module `GRAPH` defines very simple directed graphs and how to calculate reachability by finding a directed path in them.

```
fmod GRAPH is
  sorts Node Edge Graph .
  subsort Edge < Graph .

  op _->_ : Node Node -> Edge [ctor prec 40].
  op nil : -> Graph [ctor].
  op _;_ : Graph Graph -> Graph [ctor assoc comm id: nil].

  var M N O : Node .
  var G  : Graph .

  op reachable : Graph Node Node -> Bool .
  eq reachable(M -> N ; G, M, N) = true .
  ceq reachable(M -> O ; G, M, N) = true
  if reachable(G, O, N) = true .
  eq reachable(G, M, N) = false [owise].
endfm
```

The module `OID-LIST` defines a datatype for lists of `Oids`.

```
(omod OID-LIST is
  protecting DEF-OID .
  protecting LIST{Oid} * (sort List{Oid} to OidList,
                          sort NeList{Oid} to NeOidList) .
endom)
```

The module `SUBWAY-LAYOUT` defines a subway layout as a graph and provides a function to determine reachability within the layout.

```
(fmod SUBWAY-LAYOUT is
  protecting GRAPH .
  protecting OID-LIST .

  subsort Oid < Node .

  var G : Graph .
  vars St St' : Oid .
  var OL : OidList .
  op _reachable‘from_in_ : Oid OidList Graph -> Bool .
  --- It is clearly sufficient to check the first entry of the route.
  --- All the entries after the first are reachable from the first.
  --- Thus by transitivity any stations reachable from any stations
  --- after the first are reachable from the first.
  eq St reachable from St' OL in G =
     reachable(G, St':Oid, St:Oid) .
  eq St reachable from nil in G  = false .
endfm)
```

The module PASSENGER defines the sort PassengerGroup as described above and represents a group of one or more passengers going to the same location.

```
(omod PASSENGER is
  sort PassengerGroup .
  op _going‘to_ : NzNat Oid -> PassengerGroup [ctor] .
endom)

(view PassengerGroup from TRIV to PASSENGER is
  sort Elt to PassengerGroup .
endv)
```

Sets of terms of sort PassengerSet are provided by PASSENGER-SET.

```
(omod PASSENGER-SET is
  protecting SET{PassengerGroup} *
                       (sort Set{PassengerGroup} to PassengerSet,
                        sort NeSet{PassengerGroup} to NePassengerSet) .

  vars PS PS' : PassengerSet .
  vars Mz Nz : NzNat .
  vars O O' : Oid .
```

The sumpas operator sums up the total amount of passengers in a set regardless of their destinations. This is used for calculating how many passengers are on a train.

```
  op sumpas : PassengerSet -> Nat .
  eq sumpas(((Mz going to O), (Nz going to O'), PS)) =
             Mz + Nz + sumpas(PS) .
  eq sumpas(Mz going to O) = Mz .
  eq sumpas(empty) = 0 .
```

This module also defines simple operations such as addition and subtraction on these sets. If several groups in the same set are going to the same station, they simply merge and are summed into one group of passengers — e.g., if we have 2 passengers going to `st1` and 1 passenger going to `st1` we end up with 3 passengers going to `st1`.

```
  eq Nz going to O, Mz going to O = (Nz + Mz) going to O .
```

Removing passengers from a group works in a similar manner: e.g., if we have 3 passengers going to `st1` and 2 to `st2` and we wish to remove passengers going to `st1`, then we can execute the following: `remove (1 going to st1) from ((3 going to st1),(1 going to st2))`

```
  op remove_from_ : PassengerSet PassengerSet -> PassengerSet .
  eq remove empty from PS = PS .
  eq (remove ((Mz going to O),PS) from ((Nz going to O), PS')) =
     if Nz == Mz then
      empty
     else
      (sd(Nz, Mz) going to O)
     fi
      ,(remove PS from PS') .
endom)
```

The module `STATION-SCHEDULE-ENTRY` defines the sort `StationScheduleEntry` which is a pair of a set of passengers and a time when they will arrive at a station.

```
(tomod STATION-SCHEDULE-ENTRY is
  protecting PASSENGER-SET .
  sort StationScheduleEntry .
  op ps : Time PassengerSet -> StationScheduleEntry [ctor] .
endtom)

(view StationScheduleEntry from TRIV to STATION-SCHEDULE-ENTRY is
  sort Elt to StationScheduleEntry .
endv)
```

The module `STATION-SCHEDULE` extends the functionality of having one set of passengers arrive at a given time to allow us to have a list of these entries, i.e., different sets of passengers arriving at different times. This is a list of terms of the sort `StationSchedulEntry`.

```
(tomod STATION-SCHEDULE is
  protecting LIST{StationScheduleEntry} *
             (sort List{StationScheduleEntry} to StationSchedule,
              sort NeList{StationScheduleEntry} to NeStationSchedule) .
endtom)
```

The following module `SUBWAY-FUNCTIONS` introduces 2 help functions: `pickup` and `dropoff`. These are used when a train picks up or drops off passengers at a station.

```
(fmod SUBWAY-FUNCTIONS is
  protecting NAT .
  protecting OID-LIST .
```

```
protecting SUBWAY-LAYOUT .
protecting STATION-SCHEDULE .

vars M N : Nat .
vars Mz Nz : NzNat .
vars O O' : Oid .
var OL : OidList .
vars PS PS' : PassengerSet .
var G : Graph .
```

The function `dropoff` is in charge of dropping off passengers at a station and assumes the following arguments:

- An `Oid`, the name of the current station.
- An `OidSet`, the remainder of the train's given route.
- A `PassengerSet`, the passengers in the train.
- A `Graph`, the layout of the subway from the train's point of view.

The function will drop off all passengers that are going to the current station plus all the passengers going to stations unreachable from the route in the supplied subway layout.

```
op dropoff : Oid OidList PassengerSet Graph -> PassengerSet .
eq dropoff(O, OL, ((Nz going to O), PS), G) =
   (Nz going to O), dropoff(O, OL, PS, G) .
ceq dropoff(O, OL, ((Nz going to O'), PS), G) =
    if not (O' reachable from OL in G or (occurs(O',OL))) then
     (Nz going to O')
    else
     empty
    fi
    , dropoff(O, OL, PS, G)
if O =/= O' .
eq dropoff(O, OL, empty, G) = empty .
```

The function `pickup` takes the following arguments:

- A `Nat`, the number of passengers to be picked up (this may be 0).
- An `OidSet`, that represents the remainder of the train's route.
- A `PassengerSet` the passengers on the station.
- A `Graph` which represents the layout of the subway as seen from the train that calls the pickup function.

Based on the provided information, the function extracts a subset as close to the given size but not larger than the actual set of passengers at the station. Only passengers that have reachable destinations based on the route's layout will be picked for this subset.

```
op pickup : Nat OidList PassengerSet Graph -> PassengerSet .
eq pickup(Mz, OL, ((Nz going to O), PS), G) =
   if (O reachable from OL in G) or (occurs(O,OL)) then
```

```
      if Mz <= Nz then
       (Mz going to O)
      else
       (Nz going to O), pickup(sd(Mz, Nz), OL, PS, G)
      fi
     else
      pickup(Mz, OL, PS, G)
     fi .
  eq pickup(Mz, OL, empty, G) = empty .
  eq pickup(O, OL, PS, G) = empty .
endfm)
```

The following module contains the definitions needed to model trains and stations. In addition, all the rewrite rules needed to model the problem are found here.

```
(ptomod SUBWAY is
  protecting SUBWAY-FUNCTIONS .
  protecting OID-SET .
  protecting OID-LIST .
  protecting NAT-COST-DOMAIN .
  protecting NAT-TIME-DOMAIN-WITH-INF .
```

The included modules `NAT-COST-DOMAIN` and `NAT-TIME-DOMAIN-WITH-INF` set cost and time to run over the natural numbers.

We start by defining the different statuses a subway train may have.

```
  sorts StationAction TrainStatus .

  ops attached detached boarded arrived : -> StationAction [ctor] .
  op _at‘station_ : StationAction Oid -> TrainStatus .
  op traveling‘to_ : Oid -> TrainStatus .

  class Train | cars : NzNat, route : OidList, visited : OidList,
                status : TrainStatus, passengers : PassengerSet,
                activeLO : Graph, passiveLO : Graph,
                timer : TimeInf .

   class Station | trains : OidSet,
                   passengerSchedule : StationSchedule,
                   passengers : PassengerSet,
                   clock : Time .

  class SeStation | cars : Nat .
  subclass SeStation < Station .

  op traveltime : Oid Oid -> Time [comm] .
  op carcap : -> Nat .
  ops attachcost detachcost movingcost : -> Cost .
  op cars : Nat -> Nat .
  op carcost : -> Cost .

  ops allowAttach allowDetach : -> Bool .

  vars St St' Tr : Oid .
```

```
vars Ro Vi : OidList .
var OS : OidSet .
vars R Cl : Time .
var Ti : TimeInf .
vars M N : Nat .
vars Mz Nz Ca : NzNat .
vars PS PS' PS'' ON OFF STAY : PassengerSet .
var SA : StationAction .
var PSc : StationSchedule .
vars AL PL : Graph .
var PrS : PricedSystem .
var SSt : SystemState .
```

When a train is done traveling from station A to station B, the rule `arriveAtStation` changes the train's status from `traveling to` B to `arrived at station` B and puts the train's `Oid` in the set of trains present at the station. In addition, the current station is removed from the route of the train and added to the start of the train's visited list.

```
rl [arriveAtStation] :
   < Tr : Train | status : traveling to St, timer : 0,
                  visited : Vi >
   < St : Station | trains : OS >
   =>
   < Tr : Train | status : arrived at station St,
                  visited : St Vi >
   < St : Station | trains : (OS, Tr) > .
```

After a train has arrived at a service station one or more (or none) cars may be attached by applying the rule `attachCart`. This will attach an extra car provided the station has 1 or more detached cars available. Attaching cars takes no time but incurs a user-specifiable cost given as the constant `attachcost`.

```
crl [attachCart] :
    < Tr : Train | status : SA at station St, cars : Ca >
    < St : SeStation | cars : Nz >
    =>
    < Tr : Train | status : attached at station St,
                   cars : (Ca + 1) >
    < St : SeStation | cars : (Nz monus 1) >
    with cost attachcost
if allowAttach and
   (SA == arrived or SA == attached) .
```

Now all passengers that wish to disembark at this particular station do so using the function `dropoff`, while, at the same time, as many passengers that need to board and for whom the train has room for do so through the utilization of the function `pickup`. The amount of passengers that can be picked up is determined by a train's current capacity: this equals the number of cars multiplied by a user specifiable constant `carcap`. This procedure takes no time, neither does it incur any cost.

```
crl [LoadUnloadPassengers] :
    < Tr : Train | status : SA at station St, cars : Ca,
                   passengers : PS, route : Ro, activeLO : AL >
    < St : Station | passengers : PS' >
```

```
    =>
    < Tr : Train | status : boarded at station St,
                   passengers : (STAY, ON) >
    < St : Station | passengers : ((remove ON from PS'), OFF) >
if ((SA == arrived) or (SA == attached) or (SA == detached))
   /\ (OFF := dropoff(St, Ro, PS, AL)) /\ STAY := (remove OFF from PS)
   /\ (ON := pickup(((Ca * carcap) monus sumpas(STAY)), Ro, PS', AL)) .
```

After boarding is completed and if this is a service station one or more cars may be detached to save power. Provided, there are empty cars and the train consists of at least 2 cars. Detaching cars does not take time, but a user specifiable cost given by the constant `detachcost` is incurred.

```
crl [detachCart] :
    < Tr : Train | status : SA at station St, cars : Ca,
                   passengers : PS >
    < St : SeStation | cars : N >
    =>
    < Tr : Train | status : detached at station St,
                   cars : (Ca monus 1) >
    < St : SeStation | cars : (N + 1) >
    with cost detachcost
if allowDetach and Ca >= 2 and (cars(sumpas(PS)) < Ca)
   and (SA == boarded or SA == detached) .
```

Now that the train is set to pull out from a station, the rule `[trainDepart]` makes sure that the said train starts traveling towards the next station in its route. Assuming the next station in this train's route is C, C is removed from the train's route, then the status `traveling to` C is set in the train's `status` attribute. Furthermore, the departing train's timer is set using the appropriate travel time between the current station and the next.

```
crl [trainDepart] :
   < Tr : Train | status : SA at station St, route : St' Ro >
   < St : Station | trains : (OS, Tr) >
   =>
   < Tr : Train | status : traveling to St',
                  timer : traveltime(St,St'), route : Ro >
   < St : Station | trains : OS >
if SA == boarded or SA == detached .
```

The rule `tick` is a fairly standard Priced-Timed Maude tick rule, using `mte` to determine how much time can pass, `delta` to make time pass in the system, and `rate` to determine the cost per time unit elapsed.

```
crl [tick] :
    {SSt} => {delta(SSt, R)} in time R with cost (rate(SSt) * R)
if R <= mte(SSt) [nonexec] .
```

The function `delta` updates clocks for stations (stations are otherwise unaffected by the flow of time). `mte` ensures that time does not lapse past any moment passengers are scheduled to arrive.

```
eq delta(< St : Station | clock : Cl >, R)
         =
         < St : Station | clock : Cl + R > .
```

For trains, `mte` is set up to prevent trains from overshooting their destination by reading out the timer of each train. When time passes `delta`, updates the timer on every train. The rate of a train is the number of cars multiplied by a user specifiable constant `movingcost`

```
  eq mte(< St : Station | passengerSchedule : nil >) = INF .
  eq mte(< St : Station | clock : R, passengerSchedule : ps(Cl, PS) PSc >) = Cl monus R .
  eq rate(< St : Station | >) = 0 .

  --- Trains are affected by the elapse of time.
  --- The rate dependant on the number of cars
  eq mte(< Tr : Train | timer : Ti >) = Ti .
  eq delta(< Tr : Train | timer : Ti >, R)
     =
     < Tr : Train | timer : (Ti monus R) > .
  eq rate(< Tr : Train | cars : Ca >) = Ca * movingcost .
```

When passengers are due to arrive at a station they are automatically removed from the `passengerSchedule` attribute and moved into the station.

```
  eq < St : Station | clock : Cl, passengers : PS,
                      passengerSchedule : ps(Cl,PS') PSc >
     =
     < St : Station | clock : Cl, passengers : (PS, PS'),
                      passengerSchedule : PSc > .
```

The function `cars` determines how many cars are needed for a given amount of passengers.

```
  eq cars(N) = N quo carcap +
               if N rem carcap =/= 0 then 1 else 0 fi .
```

When a train has reached the end of its route (i.e. the route attribute is `nil`), the route and visited lists are switched, therefore making the train travel its previous route in reverse order.

```
  eq < Tr : Train | status : SA at station St, visited : St Vi,
                    route : nil, activeLO : AL, passiveLO : PL > =
     < Tr : Train | visited : St, route : Vi, activeLO : PL,
                    passiveLO : AL > .
```

Finally, when passengers arrive at their destination station, they are removed from the system.

```
  eq < St : Station | passengers : ((Nz going to St), PS)  >
     =
     < St : Station | passengers : PS > .
endptom)
```

Subway Execution and Analysis

We now show how to obtain the earliest and least power consuming state in which all passengers have reached their destination for the subway network depicted in Figure 7.1. Two trains will be servicing the routes covering all the stations. A first train denoted `t1` will cover `st1`, `st2`, `st3` and `st4`, i.e., route 1a/b shown in Figure 7.2 while a second train `t2` will cover the stations `st5`, `st3` and `st6`, i.e., route 2a/b shown in 7.3.

Defining an Initial State

The following module defines the subway map, travel times, routes, and the relative layouts shown in figures 7.1, 7.2, and 7.3. In addition, a suitable initial state with two trains, traveling towards the end stations and some passengers scheduled to arrive at station `st2` is defined:

```
(ptomod SUBWAY-TEST is
  protecting SUBWAY .
  protecting STRING .

  subsort String < Oid .
  ops t1 t2 : -> Configuration .
  ops st1 st2 st3 st4 st5 st6 : -> Configuration .
  ops s1 s2 s3 s4 s5 s6 : -> Oid .
  ops init1 : -> GlobalSystem .

  eq s1 = "st1" .
  eq s2 = "st2" .
  eq s3 = "st3" .
  eq s4 = "st4" .
  eq s5 = "st5" .
  eq s6 = "st6" .
```

The equations for `traveltime` define the time needed for travel between pairs of stations as shown in Figure 7.1.

```
  eq traveltime("st1", "st2") = 2 .
  eq traveltime("st2", "st3") = 3 .
  eq traveltime("st3", "st4") = 3 .
  eq traveltime("st3", "st5") = 1 .
  eq traveltime("st3", "st6") = 4 .
```

The graph `rout1a` represents the graph seen in Figure 7.2 part a. Likewise, `route1b` represents the route seen in the same figure except part c. The graph `l1uc` is the part of the subway map that is not covered by either of the two routes. When we combine the graphs `rout1a` and `l1uc` we get the relative layout seen in as Figure 7.2 part b. On the other hand combining `rout1b` and `l1uc` gives us the layout seen as part d of the same figure. The other graphs combine in a similar manner and all relate to Figure 7.2.

```
  ops route1a route1b route2a route2b l1uc l2uc : -> Graph .
  eq route1a = s1 -> s2 ; s2 -> s3 ; s3 -> s4 .
  eq route1b = s4 -> s3 ; s3 -> s2 ; s2 -> s1 .
  eq l1uc = s3 -> s5 ; s3 -> s6 .
  eq route2a = s5 -> s3 ; s3 -> s6 .
  eq route2b = s6 -> s3 ; s3 -> s5 .
  eq l2uc    = s3 -> s4 ; s3 -> s2 ; s2 -> s1 .
```

The following equations define the user specifiable parts of the system such as the capacity of each car, the power cost of moving each car, etc.

```
  eq carcap = 10 .    --- passenger capacity of each car
  eq attachcost = 5 . --- cost of attaching new car
```

```
 eq detachcost = 4 . --- cost of detaching a car
 eq movingcost = 3 . --- cost of movement per car

 eq allowAttach = false . --- disallow attaching new cars
 eq allowDetach = false . --- disallow detaching cars

 eq t1 = < "t1" : Train | cars : 3,
                          route : s2 s3 s4,
                          visited : nil,
                          status : traveling to s1,
                          timer : 0,
                          passengers : empty,
                          activeLO  : route1a ; l1uc,
                          passiveLO : route1b ; l1uc > .


eq t2 = < "t2" : Train | cars : 3,
                         route : s3 s6,
                         visited : nil,
                         status : traveling to s5,
                         timer : 0,
                         passengers : empty,
                         activeLO   : route2a ; l2uc,
                         passiveLO  : route2b ; l2uc > .


 eq st1 = < s1 : Station | trains         : empty,
                           passengers     : empty,
                           passengerSchedule : nil,
                           clock : 0 > .

 eq st2 = < s2 : Station | trains         : empty,
                           passengers     : empty,
                           passengerSchedule :
                                   ps(2,(11 going to s1,
                                          1 going to s3,
                                          1 going to s5,
                                          2 going to s6))
                                   ps(7,(2 going to s1,
                                         5 going to s3,
                                         7 going to s4,
                                         4 going to s5)),
                           clock : 0 > .

 eq st3 = < s3 : SeStation | trains         : empty,
                             cars           : 1,
                             passengers     : empty,
                             passengerSchedule : nil,
                             clock : 0 > .

 eq st4 = < s4 : Station | trains         : empty,
                           passengers     : empty,
                           passengerSchedule : nil,
                           clock : 0 > .
```

```
  eq st5 = < s5 : Station | trains         : empty,
                           passengers     : empty,
                           passengerSchedule : nil,
                           clock : 0 > .

  eq st6 = < s6 : Station | trains         : empty,
                           passengers     : 1 going to s1,
                           passengerSchedule : nil,
                           clock : 0 > .

  eq init1 = t1 t2 st1 st2 st3 st4 st5 st6 .
endptom)
```

We see from the module above that both trains are traveling towards the first station in their route. Furthermore, only the stations `st2` and `st6` will have any passengers. The station `st6` starts off with a passenger that wants to go to `st1`. On the other hand, the station `st2` does not start with any passengers, but some are scheduled to arrive at time 2 and 7.

Finding Earliest and Cheapest Solutions

We may want to know the earliest time at which all passengers have arrived at their respective destinations. The command `priced find earliest` can be used for this task:

```
Maude> (priced find earliest init1 =>*
             {< "t1" : Train | passengers : empty >
              < "t2" : Train | passengers : empty >
              < "st1" : Station | passengers : empty >
              < "st2" : Station | passengers : empty >
              < "st3" : Station | passengers : empty >
              < "st4" : Station | passengers : empty >
              < "st5" : Station | passengers : empty >
              < "st6" : Station | passengers : empty >} with not cost limit.)

Result:
{...} in time 28 with cost 504
```

This shows us that the earliest a state satisfying these conditions can be reached is at time 28.

Next we calculate the optimal amount of power consumed for bringing all passengers to their destinations by running the following search command:

```
Maude> (find cheapest init1 =>*
             {< "t1" : Train | passengers : empty >
              < "t2" : Train | passengers : empty >
              < "st1" : Station | passengers : empty >
              < "st2" : Station | passengers : empty >
              < "st3" : Station | passengers : empty >
              < "st4" : Station | passengers : empty >
              < "st5" : Station | passengers : empty >
              < "st6" : Station | passengers : empty >} with no time limit .)

Solution
....
TIME_ELAPSED:Time --> 28 ; TOTAL_COST_INCURRED:Cost --> 504
```

This shows us that the cheapest state when all passengers have reached their destinations is also the earliest state at time 28 with 504 units of power consumed.

Do Passengers Travel in The Right Direction?

We now go back to Example 7.3.1, where a lone commuter is waiting at station `st2` for a train to take him to `st4`. We now wish to determine the possibility of him getting lost on the way to his planned destination and determine if he can end up disembarking at any other station than `st2` or `st4`. for this situation, we define a suitable initial state where this passenger is the only one there is. For convenience, we list only the changes to the original initial state given the foregoing parameters and switch out the definition of station `st2` with the following:

```
eq st2 = < s2 : Station | trains         : empty,
                          passengers     : 1 going to "st4",
                          passengerSchedule : nil,
                          clock : 0 > .
```

Since this system is non terminating, or has a finite state space, we have to settle with checking whether the passenger can end up at `st4` within some reasonable time limit (in this instance, time 30). By performing the following timed search command, we can compute if this is possible:

```
Maude> (ptsearch [1] init1 =>*
             {S:SystemState
              < O:Oid : Station | passengers : 1 going to "st4" >}
              such that
              O:Oid =/= "st2" and O:Oid =/= "st4"
              in time <= 30 with no cost limit.)

No solution
```

This tells us that in time 30, our passenger will not veer off his intended course.

Chapter 8

Comparing UPPAAL CORA with Priced-Timed Maude

Price-timed systems are often specified and analyzed using *priced-timed automata* (PTA) [13]. The tool UPPAAL CORA [10] uses PTAs to model and analyze such systems. UPPAAL CORA is a priced extension of the timed automaton (TA) [25] analysis tool UPPAAL [11].

Section 8.1 gives an introduction to priced-timed automata with some simple examples. Section 8.2 shows how to specify PTA as priced-timed rewrite theories. Section 8.3 gives a short overview of UPPAAL CORA. Section 8.4 gives an example of how to specify the ETGS examples given as a Priced-Time Maude specification in Section 7.2. Finally, Section 8.5 gives a comparison between UPPAAL CORA and Priced-Timed Maude running a set of ETGS specifications on both.

8.1 A Short Overview of Priced-Timed Automata

This section defines some basics of PTAs and gives a simple PTA example.

8.1.1 Priced-Timed Automata

The PTA formalism is a fairly straightforward extension of timed automata (TA), where costs have been added to locations (delay cost) and edges (switch cost). In other words, a PTA can be seen as a graph with an associated set of clocks, where there is a cost associated with staying at a node in the graph over time and a cost associated with following an edge. Edges may be associated with conditions on the clocks and trigger clock resets.

Definition 8.1.1 (Priced-timed automata [13]) *Let X be a finite set of clocks; $\beta(X)$ the set of linear constraints over X of the form $x_i \bowtie n$, where $x_i \in X$, $n \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$; and $\beta^*(X) \subseteq \beta(X)$ such that $\bowtie \in \{<, \leq\}$. Then a PTA over the set of clocks X is a 6-tuple $A = (L, l_0, Act, E, I, \mathcal{P})$ where:*

- *L is a finite set of* locations.
- *$l_0 \in L$ is the* initial *location.*
- *Act is a finite set of* actions.
- *$E \subseteq L \times \beta(X) \times Act \times 2^X \times L$ is a set of* edges. *An edge (l, g, a, λ, l') intuitively denotes a transition from l to l' using action a; this transition also resets the clocks in λ to 0 and can only be taken if the guard g holds.*
- *$I : L \to \beta^*(X)$ assigns* invariants *to locations.*

- $\mathcal{P} : (L \cup E) \to \mathbb{N}$ *assigns* costs *to locations and edges.*

A *state* is defined as a tuple (l, u) where l is a location and u is a clock valuation. A special state is the initial state (l_0, u_0) where the PTA is in the initial location with all clocks set to zero. Transitions in a PTA can either follow an edge or wait at the current location for some time:

Definition 8.1.2 (Transitions) *There are two possible types of transitions in a PTA:* delay transitions *and* switch transitions. *A delay transition is defined as waiting for some time d in a location l. The cost associated with waiting is the location cost multiplied by the time d spent waiting. A delay transition is defined as follows:*

(1) $(l, u) \xrightarrow{d,p} (l, u + d)$ *if* $(\forall 0 \leq d' \leq d)$, $u + d'$ *satisfies* $I(l)$ *and the cost* $p = d \cdot \mathcal{P}(l)$

where $u + d' = (x_1 + d', \ldots, x_n + d')$, *i.e., adding the elapsed time d' to all clocks. Switch is the action of going from a location l to a location l' (where l and l' may be the same location) by following an edge; this can only be done if the constraints (guards) of the edge are satisfied. This will trigger whatever actions are associated with that edge, such as clock resets. The cost p of an edge transition is simply the cost associated with the edge. An edge transition is defined as follows:*

(2) $(l, u) \xrightarrow{a,p} (l', u')$ *if* $e = (l, g, a, r, l') \in E$, $u \in g, u' = u[r \to 0]$ *and cost* $p = \mathcal{P}(e)$

Notation: Looking at Figure 8.1 it is easy to sort out all the components. But to be able to avoid descriptions like the edge that goes from a to b and location a of PTA $\mathcal{A}$ some notation is needed:

- When talking about an edge going from a location l to a location l' we may refer to this edge as $e_{l,l'}$. If an edge goes from a numbered location s_n to another numbered location s_m we may also refer to this edge as $e_{n,m}$. Likewise, if the action of an edge is not given explicitly, we write $e_{l,l'}$ as the action.
- When referring to a component of a PTA, we use $PTA\text{-}name.c$ to refer to one of its components c, much like when referring to a member of objects in for instance java. For instance, the edge $e_{0.1}$ of $\mathcal{A}$ in Example 8.1.3 would be $\mathcal{A}.e_{0,1}$ likewise the location s_0 of the same PTA $\mathcal{A}.s_0$. When no ambiguity exists, we can drop the '$PTA\text{-}name.$' part.

Example 8.1.3 (PTA $\mathcal{A}$) *Figure 8.1 shows the PTA $\mathcal{A}$ with clocks x and y with locations $\{s_0, s_1, s_2\}$, initial location s_0, and edges $e_{0,1}$, $e_{0,2}$, and $e_{2,1}$. Location e_0 is the initial location and has rate 2, s_1 has cost rate 2 while s_2 has cost rate 1 and has an invariant $y \leq 4$. The edge $e_{0,1}$ costs 5 to take, resets clock y, and cannot be taken unless clock x is greater than 3.*

8.1.2 Runs, Optimal Runs, and the Mincost Reachability Problem

This section introduces the concept of a run of a PTA. A run can be seen as a sequence of transitions from the initial state (l_0, u_0). The cost of a run is the sum of the cost of the transitions.

Example 8.1.4 (Run) *A run in the PTA in Figure 8.1 could go as follows:* $(s_0, (0, 0)) \xrightarrow{4,8} (s_0, (4, 4)) \xrightarrow{e_{0,1},5} (s_1, (4, 0))$. *The cost of the run is* $8 + 5 = 13$.

A minimum cost run is the cheapest possible run from the initial state to a *goal state*. A goal state is simply a state that satisfies some properties that we can define ourselves. In the following example, we define the goal state to be any state where s_1 is the location part of the state:

Example 8.1.5 (Optimal run) *The PTA $\mathcal{A}$ may have many optimal runs, one obvious optimal run from s_0 to s_1 is* $(s_0, (0, 0)) \xrightarrow{2,4} (s_0, (2, 2)) \xrightarrow{e_{0,2},1} (s_2, (0, 2)) \xrightarrow{e_{2,1},2} (s_1, (0, 2))$. *The cost of this run is* 7.

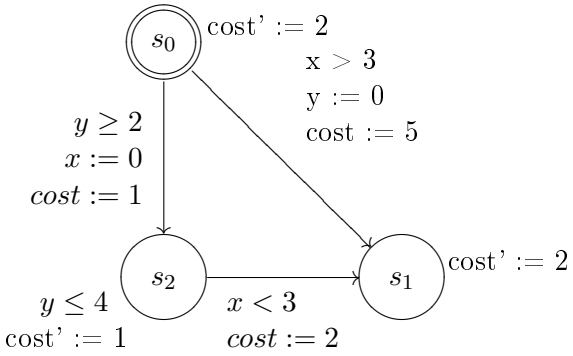


Figure 8.1: The PTA $\mathcal{A}$ from Example 8.1.3

8.2 Priced-Timed Automata as Priced-Timed Rewrite Theories

This section shows that any PTA can be translated into a priced-timed rewrite theory. Therefore, PTAs can be seen as a special case of priced-timed rewrite theories. The translation extends the translation of timed automata into real-time rewrite theories given in [20].

In the rest of this section we will assume that the set of clocks in X are ordered as $x_1, \ldots, x_n$. A PTA $A = (L, l_0, Act, E, I, \mathcal{P})$ over a set of clocks X as described in Definition 8.1.1 is represented by a priced-timed rewrite theory $\psi_{PTA}(A) = ((\Sigma_A, E_A, L_A, R_A), \varphi_A, \kappa_A, \phi_A, \tau_A)$ as follows:

- (Σ_A, E_A) contains an equational axiomatization of the time domain and cost domain[1]. Furthermore, the signature Σ_A contains a sort `Location` with a constant l of sort `Location` for each $l \in L$, a sort `PTAState` with a (n + 1)-ary operator `_,_,...,_ : Location Time ... Time` $\rightarrow$ `PTAState`, and a subsort declaration `PTAState < SystemState`.
- The set L_A of labels is the set Act of actions and one label for each delay transition $tick_l$ for each $l \in L$.
- For each $e = (l, g, a, r, l')$ in E there is a rewrite rule:

  $$\texttt{[}a\texttt{]} \ : \ \texttt{\{}l, x_1, \ldots, x_n\texttt{\}} \xrightarrow{\mathcal{P}(e_{l,l'})} \texttt{\{}l', t_1, \ldots, t_n\texttt{\}} \ \texttt{if} \ g(X)$$

  where $t_i = 0$ if $x_i \in r$ (the PTA's clock reset function) otherwise $t_i = x_i$.
- In addition, R_A contains a tick rule

  $$\texttt{[}tick_l\texttt{]} \ : \ \texttt{\{}l, x_1, \ldots, x_n\texttt{\}} \xrightarrow{y_r, \mathcal{P}(l) * y_r} \texttt{\{}l, x_1 + y_r, \ldots, x_n + y_r\texttt{\}} \ \texttt{if} \ I(l(X))$$

  for each location l in A.

Any state (l, u) in A is represented as a tuple $\texttt{\{}l, U\texttt{\}}$ in $\psi_{PTA}(A)$ where U is an n-tuple $u_1, \ldots, u_n$ of time values of sort `Time`.

Example 8.2.1 (Translating a PTA) *The PTA of Figure 8.1 can be represented by the following priced-timed rewrite theory $\psi_{PTA}(\mathcal{A})$: A set of locations: s_0, s_1, and s_2. The initial state is $\{s_0, 0, 0\}$. The next 3 rewrite rules represent the edges:*

[1] As $\mathbb{R}_+$ cannot be defined as a computable data type, we must use other time and cost domains such as $\mathbb{Q}_+$ or $\mathbb{N}_+$ that have finite axiomatizations.

$$[e_{0,1}] : \{s_0, x, y\} \xrightarrow{5} \{s_1, x, 0\} \text{ if } x > 3$$
$$[e_{0,2}] : \{s_0, x, y\} \xrightarrow{1} \{s_2, 0, y\} \text{ if } y \geq 2$$
$$[e_{2,1}] : \{s_2, x, y\} \xrightarrow{2} \{s_1, x, y\} \text{ if } x < 3$$

The following 3 rewrite rules represent waiting in each of the locations of the PTA:

$$[tick_{s_0}] : \{s_0, x, y\} \xrightarrow{z,2} \{s_0, x+z, y+z\}$$
$$[rick_{s_1}] : \{s_1, x, y\} \xrightarrow{z,2} \{s_1, x+z, y+z\}$$
$$[tick_{s_2}] : \{s_1, x, y\} \xrightarrow{z,1} \{s_2, x+z, y+z\}$$

Translating the PTA $\mathcal{A}$ into Priced-Timed Maude is now trivial:

```
(ptmod PTA-A is
 protecting POSRAT-TIME-DOMAIN .
 protecting POSRAT-COST-DOMAIN .

 sort Location PTAState .

 subsort PTAState < SystemState .

 ops s0 s1 s2 : -> Location .
 op _,_,_ : Location Time Time -> PTAState .

 vars X Y R : Time .


 crl [e01] : {s0, X, Y} => {s1, X, 0} with cost 5 if X > 3 .
 crl [e02] : {s0, X, Y} => {s2, 0, Y} with cost 1 if Y <= 2 .
 crl [e21] : {s2, X, Y} => {s1, X, Y} with cost 2 if X < 3 .

 rl [tick-s0] : {s0, X, Y} => {s0, X + R, Y + R} in time R with cost R * 2 .
 rl [tick-s1] : {s1, X, Y} => {s1, X + R, Y + R} in time R with cost R * 2 .
 rl [tick-s2] : {s2, X, Y} => {s2, X + R, Y + R} in time R with cost R .
endptm)
```

It is easy to see that α is a run in a PTA $\mathcal{A}$ if and only if $\psi_{PTA}(\alpha)$[2] is a run in $\psi_{PTA}(\mathcal{A})$. A proof sketch for this is given in [20] for timed automata and real-time rewrite theories and can be easily extended our case. This means that priced-timed rewrite theories are not at least as expressive as priced-time automata.

8.3 A Short Overview of UPPAAL CORA

The tool UPPAAL CORA [10] is an extension of UPPAAL that supports PTA with some extensions. Section 8.3.1 discusses the extensions UPPAAL CORA makes to PTA, while Section 8.3.2 gives a short overview of UPPAAL CORA specifications. Section 8.4 gives an UPPAAL CORA specification of the ETGS problem presented in Section 7.2.1.

[2]The corresponding sequence of transitions in the translation $\psi_{PTA}(\mathcal{A})$ of the PTA $\mathcal{A}$.

8.3.1 Extensions to PTA

This section gives an overview of some useful extensions that UPPAAL CORA adds to the PTA model. There are two extensions of particular interest to us: binary synchronization channels and shared variables. These extensions are implied in [13], in particular when discussing making networks of PTAs to model the ETGS problem.

Shared variables are places to store data, for instance, an integer or an array of integers. These variables may be used for the following purposes:

- Invariants on locations may evaluate any variable.
- Constraints (guards) on the edges may also include evaluating variables.
- Actions on edges are extended to allow assignment of variables.

For instance, we can have a variable n and require n to be 5 in addition to some time constraint for a specific edge to be taken. Variables are limited to types predefined by UPPAAL CORA, such as integers or characters. Arrays of shared variables may also be declared.

Binary synchronization channels are used as signals that trigger a transition for networked PTA. The label of a signal followed by `!` is used to raise a signal while edges waiting on a signal use the label followed by `?`. The following shows how this is done with a synchronization channel named *signal*:

- An edge transition with *signal*`?` will wait for *signal* to be raised. This can be seen as a constraint that can only be met by some other automata doing a *signal*`!`.
- An edge transition with *signal*`!` will raise the channel *signal*. This will in turn force *one* edge containing *signal*`?` to be taken. Note that this is something that happens simultaneously, i.e., in one step in the PTA.

To be able to talk meaningfully about a networked PTA, we need some notation:

- A network $\mathcal{A}_\mathcal{N}$ is the product of all its components i.e.: $\mathcal{A}_\mathcal{N} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$. In particular, a location l in $\mathcal{A}_\mathcal{N}$ is an element of $\mathcal{A}_1.L \times \ldots \times \mathcal{A}_n.L$.
- In addition, the transitions from definition 8.1.2 need to be extended to consider transitions taken by the whole network from one state to the next. Consider the transition $(l_i, u_i) \xrightarrow{a_i, p_i} (l_j, u_j)$. The locations l_i and l_j need to be in the product of the locations; a_i is an ordered vector of the actions taken in each of the PTAs of $\mathcal{A}_\mathcal{N}$; p_i should be the *total cost* of these actions; and u_i, u_j are valuations on the union of all the clocks of $\mathcal{A}_\mathcal{N}$.

Example 8.3.1 *Consider the network $\mathcal{A}_\mathcal{N} = \mathcal{A}_1 \times \mathcal{A}_2$ consisting of the 2 PTAs in Figure 8.2, with the initial location $l_0 = \{\mathcal{A}_1.s_0, \mathcal{A}_2.s_0\}$. Here $\mathcal{A}_1$ is set up to start $\mathcal{A}_2$ then wait for the latter to finish before terminating itself. $\mathcal{A}_2$, on the other hand, is set up to wait for $\mathcal{A}_1$ to initiate it, run for 5 time units and tell $\mathcal{A}_1$ that it is done.*

1. *$\mathcal{A}_1.e_{0,1}$ raises the signal `on` that tells $\mathcal{A}_2$ when it is not already active.*
2. *$\mathcal{A}_1.e_{1,2}$ waits for the signal `off` to be raised by $\mathcal{A}_2$.*
3. *$\mathcal{A}_2.e_{0,1}$ may be triggered by the signal `on` from $\mathcal{A}_1$, when this happens the variable active is set to 1.*
4. *$\mathcal{A}_2.e_{1,0}$ can be taken when only after 5 units of time has passed, it raises the signal `off` and sets active to 0.*

Example 8.3.2 (A run in $\mathcal{A}_\mathcal{N}$) *A typical run α in $\mathcal{A}_\mathcal{N}$ would consist of $\mathcal{A}_1$ taking edge $e_{0,1}$ at $\mathcal{A}_1.c = 0$, forcing the edge $\mathcal{A}_2.e_{0,1}$ to be taken, then waiting for 5 time units on $\mathcal{A}_2$ to switch off at $\mathcal{A}_1.c = 5$ for an accumulated cost of 26.*

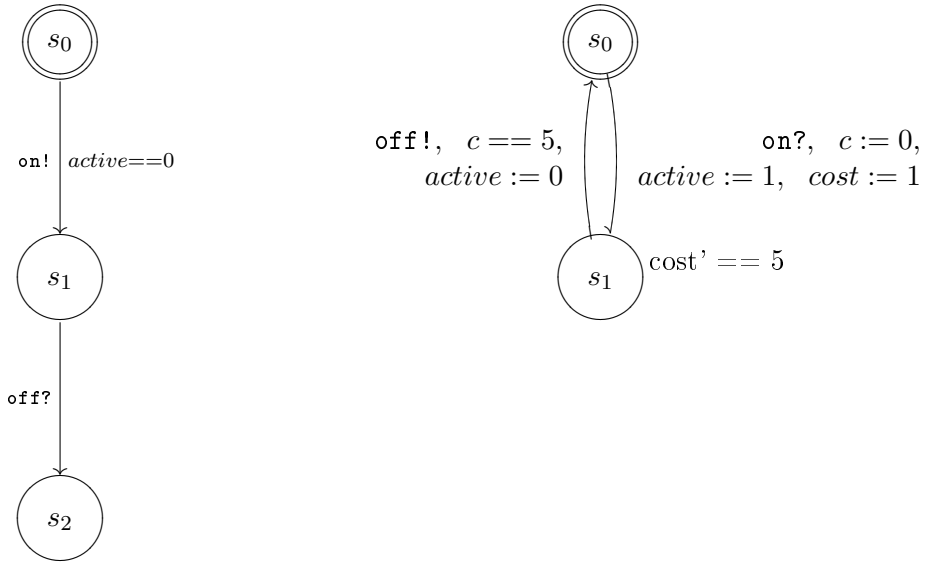


Figure 8.2: The network of 2 PTAs $\mathcal{A}_1$ (left) and $\mathcal{A}_2$ (right) described in Example 8.3.1, $\mathcal{A}_1$ switches $\mathcal{A}_2$ and waits for $\mathcal{A}_2$ to finish.

8.3.2 UPPAAL CORA Specifications

This section covers the parts of UPPAAL CORA's specification language needed to read and understand the example of the ETGS problem in the next section. This is in no way meant as a thorough introduction to UPPAAL CORA's specification language.

UPPAAL CORA provides the user with a graphical interface to specify PTAs with the extensions given in the previous section. In our example, we will be using only 3 of the specification options: *global declarations*, *templates*, and *system declarations*.

Global Declarations

Global declarations are just what they sound like: constants and variables available to the whole system. We mainly use this section to declare constants for better readability of the specification and some arrays to organize data related to our tasks and processors.

A constant is declared in a straightforward way:

`const` *datatype name* `=` *value*`;`

where *datatype* is one of the built-in UPPAAL datatypes such as `bool` or `int`. Variables are declared similarly without the equality sign and the `const` keyword.

Arrays are declared in the same way constants and variables are declared with additional size of each dimension given like the following:

datatype name`[`$size_1$`]...[`$size_n$`];`

Name: P, Parameters: const int sw, const int wt

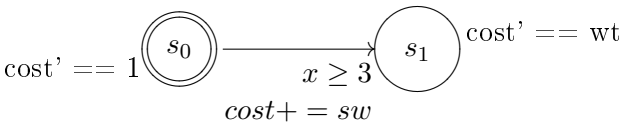


Figure 8.3: A template with the parameters sw and wt.

Templates

The *template* part of UPPAAL CORA is where PTAs are specified, i.e., the graphs representing them consist of locations, with invariants and rates; and edges with clock resets, variable and cost assignments and guards. Templates are like classes and are parametrized PTAs. We use these to avoid building a new PTA for every task and processor. When building a template, a name and parameters must be chosen. The parameters are declarations of local constants that are used in invariants, guards, assignments, and synchronization in the template.

Guards are conjunctions and disjunctions of boolean expressions of the form $a \bowtie b$, where a is a clock or a variable, b is a value, and $\bowtie$ is one of the comparison operators: $<$, $\leq$, $==$, $\geq$, $>$ or $\neq$. Multiple boolean expressions may be linked with the logical operators *and* and *or*. No guard may refer to the global cost or time.

Assignments may be clock resets that set any number of clocks to a non-negative integer including 0 as well as assign values to any number of variables or a specific cost for traversing an edge. Assignments are of the form x ?= a where x is a variable, a clock or `cost` and a is some constant and ? may be

- $+$, :, $-$ or $*$ for variables.
- : for clocks.
- $+$ for cost.

Multiple assignments can be accomplished by separating them with commas.

Synchronization may be done using binary synchronization channels on edges as described in the previous section. UPPAAL CORA supports synchronization on pairs of channels only, i.e., one *chan*`!` signal may trigger only one edge with a corresponding *chan*`?` even though there may be more edges waiting for the same signal.

Invariants are conjunctions of expressions to be evaluated on each location, like guards on edges. In addition, invariants are used to assign a rate to a location with an expression of the following form: `cost'` `==` c. The same operators and expressions used to build guards are used to build invariants. A location that does not satisfy its invariant is defined not to exist. This fact is used in 2 ways: first, when defining a rate of a location it is done as an invariant on the first order derivate of cost ensuring that locations exist with this rate; second, as time limits on reaching a location, if a location has an invariant of a clock it cannot be reached when the invariant is not satisfied as it does not exist.

Even though both cost and time runs over $\mathcal{R}_+$, all guards, invariants, and assignments of a clock must be an *integer* expression. Furthermore, assignment of rate on locations and cost on edges must also be integer expressions.

Figure 8.3 shows a template where the switch cost of the edge and the delay cost of the second location is parametrized. In the next section, we will explain how to instantiate a template.

System Declarations

System declarations are used to instantiate templates and define an initial state. This is similar to defining initial states in Priced-Timed Maude. When instantiating a template, we write: $instanceName$ = $templateName(param_1, \ldots, param_n)$ We can make an instance `test_P` of Figure 8.3 with the declaration `test_P = P(1, 3)`.

The following declaration is used to define an initial state:

`system` $instanceName_1$, ..., $instanceName_n$;

8.3.3 Solving Minimum Cost Reachability Using UPPAAL CORA

UPPAAL CORA solves the minimum cost reachability problem by internally reducing a PTA to *priced zones* [10]. Priced zones can be used to reduce an unlimited set of states to a set of states within a limited set of time constraints that share the same linear expression for rate. Linear programs can now be solved to determine which is the cheapest way to traverse a zone. This helps reduce an automaton with an unlimited set of states into a new automaton with a limited set of states. This new automaton is a quotient automaton with the priced zones as locations.

UPPAAL CORA uses a *branching and bounding*[3] algorithm to determine the best path through the quotient automata from the starting location to a goal location, which satisfies some desired criteria.

An estimate of the running time of the resulting algorithm solving the minimum cost reachability problem for the PTA A with the quotient automaton Q is as follows: $n(Z) * c^2$ where $n(Z)$ is the number of priced zones, i.e., locations in Q, the size of this number depends on how A can be reduced to zones but is generally a linear function of the clock constraints and locations of A. The variable c is the number of clock constraints on each priced zone in A. For each priced zone a linear program has to be solved over the clock constraints to find the cheapest possible way to traverse a zone. The simplex algorithm is used for solving these linear programs, this algorithm has running time of $O(2^n)$ (often in practical examples it has been shown to run in $O(n * log(n))$). This estimate is based on the algorithm presented in [10].

8.4 Example: Modeling ETGS in UPPAAL CORA

To contrast how the ETGS problem is modeled using Priced-Timed Maude and UPPAAL CORA, this section shows how the ETGS problem *may* be modeled using UPPAAL CORA. The specifications and techniques used for further testing are based on the PTA specification techniques given in [13]. The techniques from [13] are modified with regard to making the templates for tasks more flexible so that each task does not need its own template. We use the ETGS from Example 7.2.3, with three tasks, two processors, and one bus.

We define one template for processors and one for tasks. Each processor and the bus[4] are instantiated with the processor template using their distinct properties as arguments. Each of the tasks, on the other hand, is instantiated with the task template with the individual task's data as arguments. In order to accomplish this, we store all the information relevant to processor power consumption in two arrays that are named `pi` and `tau`. Task execution and broadcast times are stored in a third array called `delta`. The specification contains the following elements:

- 2 PTA templates: one for tasks and one for processors.

[3] Branch and bound [26] is a general algorithm for finding optimal solutions of various optimization problems. It consists of a systematic enumeration of all candidate solutions, where branches of the search tree that contain fruitless candidates are eliminated, by using upper and lower estimated bounds of the quantity being optimized. The efficiency of a branch and bound algorithm depends heavily on the bounding function.

[4] The bus works exactly like a processor, but with different name for broadcast time, i.e. κ for broadcast time instead of δ for running time.

- A vector `act` of 3 booleans keeping track of the active status of the bus and the two processors.
- A vector `d` of 3 keeping track of the current remaining run time for a task on a processor or bus (this cannot be stored with each processor).
- Vectors `pi` and `tau` of 3 integers each, keeping track of the idle and active rates of the bus and the processors.
- A 3 by 3 array `res` keeping track of which processor knows the result of what task.
- A 3 by 3 array `delta` keeping track of the running times of each task on each processor and the bus.
- A 3 by 2 array `pre` holding the predecessor information for each task.
- An array `pbus` containing the signals for each processor and the bus.

Note that all the above arrays and vectors need to be global as otherwise networked PTAs may not manipulate them. This means that almost all data is stored in global variables and nothing is passed between PTAs in a way one would expect it to be in an object-oriented system.

8.4.1 Global Declarations

The following UPPAAL CORA specification models the ETGS problem in Example 7.2.3: We start by declaring some constants that state how many tasks and processors are running in the system. These will be used when declaring the different arrays later on.

```
const int tasks = 3;
const int procs = 2;
```

The next constants are used to make the specification more readable: they give the subscripts of the bus, the processors, and the different tasks in the arrays relevant to them, e.g., typing `arrayName[p1]` is the same as typing `arrayName[1]`.

```
const int bus = 0;          const int t1 = 0;
const int p1 = 1;           const int t2 = 1;
const int p2 = 2;           const int t3 = 2;
```

The `act` array keeps track of which processors are currently active, the element with subscript 0 of this array is the bus.

```
bool act[procs + 1];
```

The `d` array keeps track of remaining execution times on the different processors. The element with subscript 0 of this array is the bus and denotes remaining broadcast time for the task currently being broadcast on the bus.

```
int d[procs + 1];
```

The `res` array keeps track of which processors know the result of which tasks, e.g., `res[p1][t1] := true` means the processor `p1` knows the result of task `t1`.

```
bool res[procs + 1][tasks];
```

The next array pbus is that of synchronization channels. These signals are used to activate processors and the bus that a task is done processing or broadcasting from a processor or bus.

```
chan pbus[procs + 1];
```

The next two arrays pi and tau give the active and idle rates of the bus and the processors, e.g., tau[bus] equals 11, this tells us the active rate of the bus is 11.

```
const int pi[procs + 1] = 11,5,4;
const int tau[procs + 1] = 1,1,1;
```

The array delta holds information about running time for each task on each processor, the first column denotes broadcast times, the second running times on p1, and the third denotes running times on p2. The constant na means the task cannot run on the relevant processor, e.g., from the array we see thatt1 cannot be run on p2. In the code below we use comments that start with // and comment out the rest of the line to make the code more readable.

```
const int na = -1;
const int delta[tasks][procs +1] = {{ 7,  1, na},  //t1
                                    { 5, na,  2},  //t2
                                    { 6,  5,  4}}; //t3
```

The array pre gives the dependencies of a task, e.g., task t1 depends on the results of no other tasks while the task t3 depends on the result of both t1 and t2.

```
const int none = -1;
const int pre[tasks][2] =   {{none, none},  //t1
                             {none, none},  //t2
                             {  t1,   t2}}; //t3
```

We define a global clock systemClock. This clock is used to make sure the deadline on a schedule is satisfied.

```
clock systemClock;
```

Finally, the global deadline for a feasible schedule is given as the constant dl.

```
const in dl = 12;
```

8.4.2 The Processor Template

The processor/bus automaton template is shown in Figure . The top part shows the name and the parameters for this template. The parameters are to be used as follows:

- p is the identifier for this processor, i.e., bus, p1, or p2.
- tau is the idle rate of the processor, this is looked up in the array tau on instantiation of the template, e.g., for the bus this is tau[bus].
- pi this is the active rate of the processor.

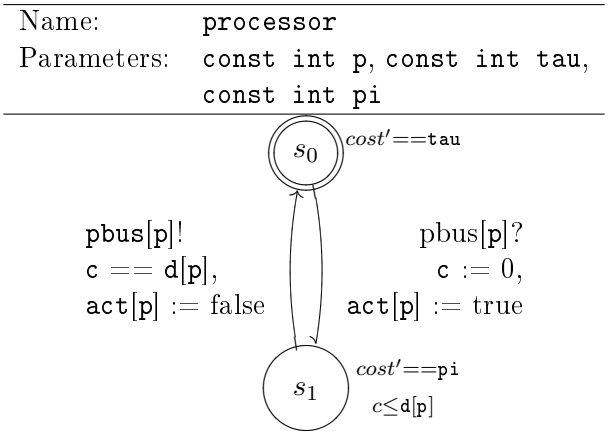


Figure 8.4: Template for processor/bus automata.

In addition to the parameters of the template, a local clock `c` is declared. If we want to use this template to instantiate our bus we can use the following declaration: `proc_bus = processor(bus, tau[bus], pi[bus])`, the resulting PTA `proc_bus` will now have the properties of the bus with active rate 11 and idle rate 1.

We will now look at how exactly this automata works. The automaton consists of 2 locations: the initial location s_0 the idle an state of the processor and s_1 the active state of the processor. In addition, there are two edges: $e_{0,1}$ activates the processor and starts processing the current task and $e_{1,0}$ deactivates the processor when the current task is finished.

The processor may stay in the idle state indefinitely and incur cost at a rate equal to the *tau* parameter. However, when a task raises the processors synchronization signal, this activates the processor and makes it traverse the edge $e_{0,1}$. For instance, some task may raise the signal `pbus[p1]!`, forcing the automaton representing `p1` to take the edge $e_{0,1}$.

When the edge $e_{0,1}$ is traversed, the internal clock `c` is set to 0 and the appropriate record in the `act` array is set to `true`, flagging the processor as active. For example, if `p1` is activated, its clock is set to 0 and `act[p1] := true`.

From here the automaton switches to the location s_1, its active state. Once the automaton is in this state, it must stay there until its internal clock reaches the running time of the task that started it. At the same time, cost is incurred at the rate given by the parameter `pi`. For instance, if this is the automaton representing `p1` and it was started by `task1` it would have to stay in the active state for `delta[t1][p1]` (1) time units while adding 5 cost to the system every time unit.

Once the automaton is done with running its current task, it switches to the edge $e_{1,0}$ where the `act` record for the processor is set to `false`indicating the processor is inactive. Furthermore, a signal is sent back to the task that started the processor to indicate that the processing is complete. For instance, if `p1` is done processing the task `t1`, the signal `pbus[p1]!` is raised and `act[p1]` is set to false.

Finally, the automaton reverts to the idle state, s_0 where it may stay until it is signaled to start again.

8.4.3 The Task Template

The task template is named `task2` and is shown in Figure 8.5. The parameters for a task automaton are as follows:

- `task`: this is the name of the task, i.e., `t1`, `t2`, or `t3`.
- `pre1` and `pre2` are names of tasks the task depends on, e.g., task `t3` depends on `t1` and `t2`. If a task depends on 1 or 0 other tasks, one or both of these will be the value `none`.

- `delta1` and `delta2` is the running time on `p1` and `p2`, e.g., the task `t1` has running time 1 on `p1` and cannot run on `p2`, therefore, `delta1` for `t1` is 1 and `delta2` is `na`. `delta1` and `delta2` are looked up in the array `delta` upon instantiation, i.e., `t1`'s `delta1` and `delta2` are `delta[t1][p1]` and `delta[t1][p2]`.
- `kappa`: this is the time required to broadcast the task on the bus.

This template is built for up to 2 processors (indicated by the name `task2`) and check on up to 2 predecessors. It has 6 locations and 6 edges: s_0 is the initial location where a task is waiting to start processing.

The edges $e_{0,1a}$ and $e_{0,1b}$ initiate processing either on `p1` or `p2` if the task is able to run on the appropriate processor. This means that this particular processor is not already active and there is a running time associated with it. In addition, said processor must know the results of relevant predecessors of the task.

Locations s_{1a} and s_{1b} are the locations where a task is being processed on `p1` and `p2`, respectively. When a task is finished processing either of the two processors, the result is set in that processor's result element in the `res` array.

Location s_2 is the done state. This state must be reached within the global deadline `dl` to meet a schedule's deadline. From here the task may be broadcast whenever the bus is free. The edge $e_{2,3}$ initiates a broadcast on the bus when it is not already busy. In location s_3, the task waits for the bus to finish broadcasting. When the bus is done broadcasting, the edge $e_{3,4}$ sets the result as known in both processor `p1` and `p2`. The task then switches to its final state s_4 where it will stay indefinitely.

8.4.4 System Declarations and Initial State

We instantiate the 6 different PTAs using the two templates: First, we instantiate the PTA for task `t1`:

```
task_t1 = task2(t1, pre[t1][0], pre[t1][1], delta[t1][p1], delta[t1][p2], delta[t1][bus]);
```

This line says that the name of the PTA will be `task_t1`, the template used will be the template `task2`. Moreover, the parameters that are passed to the template tell us that the name of the task is `t1`, predecessors are found in `pre[t1][0]` and 1, running times `delta1` and `delta2` are found in `delta[t1][p1]` and `delta[t1][p2]`, and the broadcast time `kappa` is found in `delta[t1][bus]`. The rest of the task PTAs are instantiated in the same manner.

```
task_t2 = task2(t2, pre[t2][0], pre[t2][1], delta[t2][p1], delta[t2][p2], delta[t2][bus]);
task_t3 = task2(t3, pre[t3][0], pre[t3][1], delta[t3][p1], delta[t3][p2], delta[t3][bus]);
```

The PTAs for the bus and the two processors are instantiated in a similar manner. The following says that the bus is a PTA named `proc_bus` using the template `processor` with the name `bus` with active rate `tau[bus]` and idle rate `pi[bus]`. This is done analogously for the 2 processors.

```
proc_bus = processor(bus,tau[bus],pi[bus]);
proc_p1 = processor(p1,tau[p1],pi[p1]);
proc_p2 = processor(p2,tau[p2],pi[p2]);
```

Finally, we must specify what PTAs are used in the running system. The following declares that the system will be made up of the 6 PTAs we just instantiated:

```
system proc_bus, proc_p1, proc_p2, task_t1, task_t2, task_t3;
```

8.5 Performance Comparison Between UPPAAL CORA vs Priced-Timed Maude: Energy Task Graph Scheduling

In this section, we show the results of executing a series of tests using the ETGS specification from Chapter 7 and UPPAAL CORA specification of the same problem set.

The tests include measuring how fast both applications can obtain the optimal schedule for problems using their built-in capabilities for this. In addition, we used the regular search command in Price-Timed Maude to obtain a feasible schedule for each of the configurations. The configuration used are as follows: 3, 5, 7, 9 and 10 tasks while using 2, 3, and 4 processors in addition to the bus.

All UPPAAL CORA tests were made using the command line tool `verifyta` supplied with UPPAAL CORA. The option `-E` was used to obtain the optimal schedules. The feasible schedules were obtained using default settings (breadth first search, default space reduction and no trace printing). For both cases average running times were obtained using a python script that ran each test for 5 iterations then calculated the average. All Priced-Timed Maude tests were done by using the command `find cheapest` with appropriate time bounds to obtain the optimal schedules and `ptsearch` with the appropriate time bound to obtain feasible schedules. Times were obtained by recording the rewrite time from Maude during 5 runs and computing the average. All tests were run on an Intel Core 2 Duo 2.2 GHz with 2 GB of memory.

The following table shows the results in number of seconds:

App	P	Method	3	5	7	9	10
CORA	2	Optimal	0.1	0.1	0.1	0.1	1.0
		Feasible	0.1	0.1	0.1	0.4	0.7
PTM	2	Optimal	1.1	3.5	8.9	185.2	289.1
		Feasible	0.4	1.7	4.7	103.9	168.6
CORA	3	Optimal	0.1	0.1	0.2	0.2	0.3
		Feasible	0.1	0.1	0.1	0.4	0.6
PTM	3	Optimal	1.5	5.5	17.0	360.4	521.5
		Feasible	0.5	3.1	9.6	210.0	339.7
CORA	4	Optimal	0.2	0.2	0.3	0.8	1.0
		Feasible	0.1	0.1	0.1	0.4	0.6
PTM	4	Optimal	7.6	15.3	94.8	—	—
		Feasible	1.5	10.7	38.5	—	—

The entries in the column named 'App' denote whether Priced-Timed Maude or UPPAAL CORA was used. The second column says how many processors were used in the specification. The third column says whether a normal search for a feasible schedule or a search for an optimal schedule was performed. The headings of columns 4 to 6 denote how many tasks were used in the specification. For instance, we can see that the entry in row 3, column 4 was a search for an optimal solution using Priced-Timed Maude, the specification contained 2 processors and 3 tasks and the running time was 1.1 seconds. The entries marked '—' failed to terminate in a reasonable amount of time due to memory usage, resulting in a swapping to disk situation.

As seen, UPPAAL CORA solves this set of problems significantly faster than Priced-Timed Maude. The reduction of PTA into priced zones in UPPAAL CORA is based in a large part on the time constraints specified in the model. The maximum amount of time constraints at any given time in the UPPAAL CORA specification of these ETGS test cases will never exceed the number of processors plus one (for the bus), therefore, they will be reduced to a relatively small quotient automaton that can be solved efficiently.

Priced-Timed Maude, on the other hand, does not employ any such reduction techniques. The running time of the algorithm could, in the worst case scenario, be $O(n^x)$ where n is dependent upon the number of rules and processors and x is dependent on the number of tasks and the time sampling strategy. The current implementation of the `find cheapest` command executes `ptsearch` successively and does not

employ any reduction methods, therefore, running `ptsearch` to obtain a feasible solution is naturally faster.

This behavior is as expected, UPPAAL CORA is founded on the relatively simple PTA model relying on automata theory that has successfully been analyzed using linear programming. Priced-Timed Maude, on the other hand, is a programming language supporting richer and more flexible specification techniques, like user-definable data types, object-oriented specification, and advanced communication features. In Section 8.2 we show that any PTA can be translated to Priced-Time Maude, while the opposite is not true. The priced thermostat from Section 4.5.1 is a simple example of a Priced-Timed Maude specification that can not be represented as a PTA as UPPAAL CORA does not allow specification of a rate on variables in locations apart from on the global cost.

Name:	task2
Parameters:	const int task, const int pre1, const int pre2, const int delta1, const int delta2, const int kappa

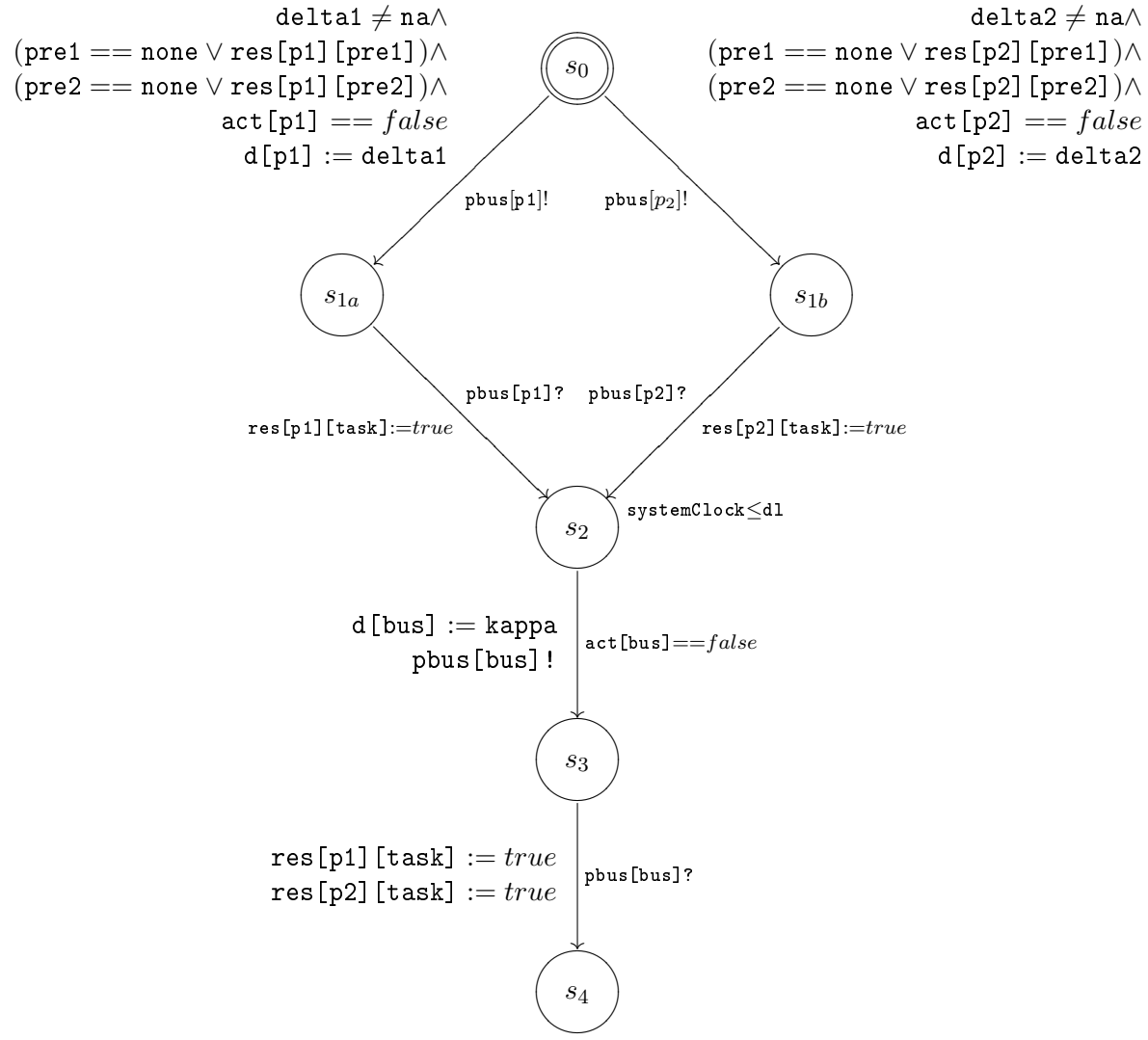


Figure 8.5: Template for the task automata.

Chapter 9

Concluding Remarks

This thesis has investigated the suitability of using Priced-Timed Maude for the specification and analysis of priced-timed systems. In particular, it has investigated the suitability of modeling such systems in an object-oriented style. I have done this by defining two theoretical models: *priced* and *priced-timed rewrite theories*. In addition, a tool, Priced-Timed Maude, has been developed to specify and analyze such theories. Priced-Timed Maude has been built in Maude as an extension of Real-Time Maude.

The results of the investigation have been partially positive. I have developed intuitive specification techniques for the specification of priced-timed systems that extend the corresponding techniques in Real-Time Maude. In particular, Priced-Timed Maude supports the large and important class of "flat" *object-oriented* priced-timed systems. The main value of Priced-Timed Maude lies in its ability for intuitive specification and analysis of such flat object-oriented systems. Priced-timed rewrite theories have, unsurprisingly, been shown to be more expressive than priced-timed automata (PTA) by giving a translation from priced-timed automata into priced-timed rewrite theories, and by giving one example that cannot be modeled using PTAs but that has been modeled in Priced-Timed Maude.

Priced-Timed Maude has successfully been applied to some larger case studies. The tool was used to model the *airplane landing problem* (ALP) and to obtain an optimal solution for a system with two airplanes and a runway. The *energy task graph scheduling* (ETGS) problem was modeled and, in addition, an optimal schedule was obtained for a system with three tasks, two processors, and a bus. The *subway passenger routing* (SPR) problem was modeled using a subway network with two internal routes, two trains, and some passengers. Among other things, we determined that a passenger cannot end up farther from the destination than he started if he follows the rules of the system. In addition to specifying and analyzing these systems in Priced-Timed Maude, I have modeled and analyzed one of the examples, ETGS, in Uppaal CORA for a comparison of performance and specification language.

When comparing the performance for the ETGS problem in Priced-Timed Maude and Uppaal CORA, Priced-Timed Maude was slower by a large margin. This is expected, as Priced-Timed Maude is a more general tool than Uppaal CORA and can be used to specify and analyze a much larger class of systems. Furthermore, Uppaal CORA is based on PTA that has an established theory for reducing the state space of an automaton using priced zones, thereby reducing the problem to a linear programming problem within each zone. As of now, there exists no "reduction" theory for priced-timed rewrite theories.

The current prototype of the tool may only be used to specify and analyze flat and flat object-oriented systems even though priced-timed rewrite theories have no such restriction. This is an implementation issue and a calculated tradeoff as time was limited for implementing the tool during this thesis. Nevertheless, flat and flat object-oriented priced-timed systems represent large enough classes of systems to justify this choice, both because all examples of priced systems encountered so far have been possible to model within these restrictions and because all major Real-Time Maude applications [4, 5, 6, 7, 8, 9] have been modeled as flat object-oriented systems.

Future Work

I would like to see better performance of the tool when solving general reachability problems as well as optimization problems. The solution to this may be to develop a reduction theory for priced-timed rewrite theories in combination with tweaking the implementation of the tool. Developing reduction theory for the general case may not be possible. In this case, developing specification techniques and reduction theory for specific classes of priced-timed rewrite theories and solving these more efficiently may be preferable.

Finally, allowing non-flat systems to be specified and analyzed by Priced-Timed Maude should be implemented. Although all interesting systems we have seen so far have been able to fit nicely into the flat object-oriented class, some systems that do not fit intuitively into this class may be encountered in the future.

Bibliography

[1] P. C. Ölveczky and J. Meseguer. The Real-Time Maude tool. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS'08*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[2] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2007.

[5] M. Katelman, J. Meseguer, and J. Hou. Formal modeling, analysis, and debugging of a wireless sensor network protocol with Real-Time Maude and statistical model checking. Technical report, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2008. In preparation.

[6] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29(3):253–293, 2006.

[7] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.

[8] E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, 2004.

[9] P. C. Ölveczky, P. Prabhakar, and X. Liu. Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In *IPDPS'08*. IEEE, 2008.

[10] Gerd Behrmann. UPPAAL CORA home page: `http://www.cs.auc.dk/~behrmann/cora/index.html`, 2006.

[11] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[12] M. Wen, J. Larsen, and J. Clausen. An exact algorithm for aircraft landing problem. Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, 2005.

[13] J. I. Rasmussen, K. G. Larsen, and K. Subramani. On using priced timed automata to achieve optimal scheduling. *Form. Methods Syst. Des.*, 29(1):97–114, 2006.

[14] Kim Guldstrand Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505, London, UK, 2001. Springer.

[15] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[16] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[17] N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285, 2002.

[18] U. Waldman. Semantics of order-sorted specifications. *Theoretical Computer Science*, 94:1–35, 1992.

[19] P. C. Ölveczky. Formal modeling and analysis of distributed systems in Maude. Course book for INF3230, Dept. of Informatics, University of Oslo, 2008.

[20] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[21] P. C. Ölveczky and M. Grimeland. Formal analysis of time-dependent cryptographic protocols in Real-Time Maude. In *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE Computer Society Press, 2007.

[22] F. Durán and P. C. Ölveczky. A guide to extending Full Maude illustrated with the implementation of Real-Time Maude. In *Seventh International Workshop on Rewriting Logic and its Applications (WRLA'08)*, 2008. To appear.

[23] P. C. Ölveczky. *Real-Time Maude 2.3 Manual*, 2007. `http://www.ifi.uio.no/RealTimeMaude/`.

[24] Alexander Schrijver. *A Course in Combinatorial Optimization*. Schrijver, Alexander, 2006.

[25] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[26] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

Appendix A

Code for Running Examples

This appendix lists all example code used in this thesis.

A.1 Example Priced-Timed Maude Specifications

This section lists all code for the example specifications given in Chapter 4.

```
(ptmod PRICED-THERMOSTAT is

  protecting POSRAT-TIME-DOMAIN .
  protecting POSRAT-COST-DOMAIN .

  sort Status Thermostat .
  subsort Thermostat < SystemState .

  ops on off : -> Status [ctor] .
  op _`,_ : Status PosRat -> Thermostat [ctor] .

  rl [turn-on] :   off , 62  =>  on , 62  with cost 50 .
  rl [turn-off] :  on , 74  =>  off , 74 .

  vars R R' : Time .
  var T : PosRat .

  crl [tick-on] :
      {(on, T)} => {(on, T + (2 * R'))} in time R' with cost R' * 100
  if R' <= ((74 - T) / 2) [nonexec] .

  crl [tick-off] :
      {(off, T)} => {(off, T - R')} in time R' if R' <= (T - 62) [nonexec] .
 endptm)


(ptmod MODEL-CHECK-THERMOSTAT is
  protecting PRICED-MODEL-CHECKER .
  protecting PRICED-THERMOSTAT .

  ops therm-on therm-off : -> Prop [ctor] .
  op temp-is : PosRat -> Prop [ctor] .

  vars T T' : PosRat .
  var S : Status .

  eq {off, T} |= therm-off = true .
  eq {on, T} |= therm-on = true .
  eq {S, T} |= temp-is(T') = (T == T') .
 endptm)


(ptomod PRICED-TIMED-OO-LIGHT-SWITCH is
  protecting NAT-TIME-DOMAIN-WITH-INF .
  protecting NAT-COST-DOMAIN .

  sort Status .
  ops on off : -> Status [ctor] .

  class Switch | status : Status, wattage : Cost, timer : TimeInf  .

  var O : Oid .
  var W : Cost .
```

```
    var Ti : TimeInf .
    vars T R : Time .
    var S : SystemState .

    rl [turn-off] : < O : Switch | status : on, timer : 0 >
                    =>
                    < O : Switch | status : off, timer : INF > .

    rl [turn-on] : < O : Switch | status : off, wattage : W, timer : INF >
                   =>
                   < O : Switch | status : on, timer : 5 > with cost W .

    eq mte(< O : Switch | status : on, timer : Ti >) = Ti .
    eq mte(< O : Switch | status : off >) = INF .

    eq delta(< O : Switch | status : on, timer : Ti >, R) =
             < O : Switch | timer : Ti monus R > .
    eq delta(< O : Switch | status : off >, R) =
             < O : Switch | > .

    eq rate(< O : Switch | status : on, wattage : W >) = W .
    eq rate(< O : Switch | status : off >) = 0 .

    crl [tick] : {S} => {delta(S, R)} in time R with cost (rate(S) * R)
                 if R <= mte(S) [nonexec] .
  endptom)


  (ptomod TEST-TWO-LIGHTS is
    protecting PRICED-TIMED-OO-LIGHT-SWITCH .
    protecting STRING .

    subsort String < Oid .

    ops init1 init2 : -> GlobalSystem .
    ops driveway garden : -> Configuration .

    eq driveway = < "Driveway" : Switch | status : off, wattage : 40, timer : INF > .
    eq garden = < "Garden" : Switch | status : off, wattage : 25, timer : INF > .
    eq init1 = {driveway} .
    eq init2 = {driveway garden} .
  endptom)
```

A.2 Example Priced-Timed Maude Analysis

This section lists all the analysis and results from Chapter 5.

```
(ptfrew init1 in time <= 15 with no cost limit.)

Result PricedTimedSystem :
  {< "Driveway" : Switch | status : on, timer : 2, wattage : 40 >} in time 15 with cost 640


(ptfrew {off, 62}  with no time limit with cost <= 6000 .)

Result PricedTimedSystem :
  {on,64} in time 163 with cost 6000


(ptsearch [1] {off, 62} =>* {off, P:PosRat} such that P:PostRat < 62 with no limits.)

No more solutions


(ptsearch [1] {off, 62} =>* {S:Status, 68} in time <= 5 with no cost limit.)

Solution 1
S:Status --> on ; TIME_ELAPSED:Time --> 3 ; TOTAL_COST_INCURRED:Cost --> 350

No more solutions


(ptsearch [1] {< "Driveway" : Switch | status : on, wattage : 40, timer : 5 >}
              =>*
              {< "Driveway" : Switch | status : off >} in time <= 2 with no cost limit.)

No more solutions


(find cheapest {off, 62} =>* {S:Status, 70} with no time limit .)

Solution
S:Status --> on ; TIME_ELAPSED:Time --> 4 ; TOTAL_COST_INCURRED:Cost --> 450
```

```
(find cheapest {< "Driveway" : Switch | status : on, wattage : 40, timer : 5 >}
               =>*
               {< "Driveway" : Switch | status : off >} with no time limit .)

Solution
CLASS_OF_"Driveway":Switch --> Switch ;
REMAINING_ATTRIBUTES_OF_"Driveway":AttributeSet --> timer : INF, wattage : 40  ;
TIME_ELAPSED:Time --> 5 ; TOTAL_COST_INCURRED:Cost --> 200


(priced find earliest  {< "Driveway" : Switch | status : on, wattage : 40, timer : 5 >}
                       =>*
                       {< "Driveway" : Switch | status : off >} with no cost limit.)

Result: {< "Driveway" : Switch | status : off, wattage : 40, timer : INF >} in time 5 with cost 200


(priced find earliest  {off, 62} =>* {S:Status, 70} with no cost limit.)

Result: {on,70} in time 4 with cost 450


(pmc {off, 62} |=u <> therm-on .)

Result Bool:
true


(pmc {on, 62} |=u therm-on U therm-is(72) .)

Result Bool:
true
```

A.3 Case Study Specification and Analysis

This section lists all the code and analysis results for each of the case studies in Chapter 7.

A.3.1 ALP

```
  sorts PlaneType PlaneTypeNone .
  subsort PlaneType < PlaneTypeNone .

  ops big small medium : -> PlaneType [ctor] .
  op None : -> PlaneTypeNone [ctor] .

  class Plane | type : PlaneType, landingTime : TimeInf,
                landAt : DefOid, earliest : Time, target : Time,
                latest : Time, early : Cost, late : Cost,
                latePenalty : Cost, clock : Time, rate : Cost .

  class Runway | landed : OidTimeList, type : PlaneTypeNone,
                 prevStart : Time .

  var P RW : Oid .
  var OTL : OidTimeList .
  vars C C1 ER : Cost .
  var TI : TimeInf .
  var R T L E Lt Ta : Time .
  var PT : PlaneType .
  var PTN : PlaneTypeNone .
  var SSt : SystemState .

  op sep : PlaneType PlaneTypeNone -> Nat .

  eq sep(PT, None)       = 0 .
  eq sep(small, big)     = 3 .
  eq sep(medium, big)    = 2 .
  eq sep(big, big)       = 1 .
  eq sep(small, medium)  = 2 .
  eq sep(medium, medium) = 1 .
  eq sep(big, medium)    = 1 .
  eq sep(PT, small)      = 1 .

 crl [assignToNextFree] :
      < P : Plane | type : PT, earliest : E, landAt : noOid, latest : Lt,
                    clock : R, late : LR, early : ER, target : Ta >
      < RW : Runway | type : PTN, prevStart : T >
      =>
      < P : Plane | landAt : RW, landingTime : L, rate : C >
      < RW : Runway | type : PT, prevStart : L >

      with cost (if L == R and L =/= E then
                 totEarlyCost(Ta, L, E, ER)
                else
```

```
             0
           fi)
 if L :=
         if R >= (T + sep(PT, PTN)) then
            R
         else
            T + sep(PT, PTN)
         fi


    /\
    C := (if R < Ta and R >= E and L < Ta then
          --- accelerated rate equals total cost from earliness
          --- / the time interval left to accelerate
          (totEarlyCost(Ta, L, E, ER) / (Ta monus R))
         else
          0
         fi)
   /\ L >= E  /\ L <= Lt .


 crl [assignToTarget] :
     < P : Plane | type : PT, latePenalty : C, earliest : E,
                   landAt : noOid, latest : Lt, target : Ta,
                   clock : R >
     < RW : Runway | type : PTN, prevStart : T >
     =>
     < P : Plane | landAt : RW, landingTime : Ta >
     < RW : Runway | type : PT, prevStart : Ta >
 if Ta >= (T + sep(PT, PTN)) .

rl [planeLanding] :
   < P : Plane | landAt : RW, landingTime : L, clock : L,
                 earliest : E, target : Ta, early : C, latePenalty : C1 >
   < RW : Runway | landed : OTL >
   =>
   < RW : Runway | landed : (OTL (P, L)) >
   with cost
   (if Ta < L then C1 else free fi
    pluss
 if L == E and Ta =/= E then C else free fi) .


 crl [tick] :
     {SSt} => {delta(SSt, R)} in time R with cost (rate(SSt) * R)
  if R <= mte(SSt) [nonexec] .

 --- Runways do not generate cost or affect time
 eq delta(< RW : Runway | >, R:Time)
    =
    < RW : Runway | > .
 eq mte(< RW : Runway | >) = INF .
 eq rate(< RW : Runway | >) = 0 .


 eq mte(< P : Plane | landAt : noOid, earliest : E, target : Ta,
                      clock : R, latest : L >) = if R < Ta then
                                                  Ta monus R
                                                 else
                                                  L monus R
                                                 fi .

 --- any assigned plane before earliest
 ceq mte(< P : Plane | landAt : RW, clock : R, landingTime : L, earliest : E >)
     = E monus R
  if R < E .

 --- landing time
 eq mte(< P : Plane | landAt : RW, clock : L, landingTime : L >) = 0 .

 --- landing on target
 eq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : T >) = T monus R .

 --- early before target
 ceq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : L, earliest : E >)
     =  L monus R
  if E < T /\ R < T  /\ R >= E .

 --- late between earliest and target
 ceq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : L, earliest : E >)
     = T monus R
  if L > T /\ R >= E /\ R < T .

 --- late between target and landing time
 ceq mte(< P : Plane | landAt : RW, clock : R, target : T, landingTime : L >)
      = L monus T
  if L > T /\ R >= T .

 eq delta(< P : Plane | clock : L >, R)
    =
    < P : Plane | clock : L plus R > .

 eq delta(< P : Plane | clock : L >, R)
    =
```

```
    < P : Plane | clock : L plus R > .

 --- If time is before earliest there's no rate
 ceq rate(< P : Plane | earliest : E, clock : R >) = 0
  if R < E .

 --- If plane is scheduled to be on target there's no rate
 ceq rate(< P : Plane | clock : R, landingTime : L, target : L >) =  0 if R < L .

 --- if target time has been reached the rate is late
 ceq rate(< P : Plane | late : C, target : Ta, clock : R >) = C
  if R >= Ta .

 ceq rate(< P : Plane | landAt : RW, earliest : E, early : ER,
                        target : Ta, landingTime : L, clock : R, rate : C >)
     =  if C == 0 then
        --- cost per time unit equals the total early cost divided by the
        --- interval between target and landing
        (totEarlyCost(Ta, L, E, ER) / (Ta monus L))
        else
         C
        fi
  if R >= E and L < Ta and R =/= Ta .

 op totEarlyCost : Time Time Time Cost -> Cost .
 eq totEarlyCost(Ta, L, E, ER) = ((Ta monus L) / (Ta monus E)) * ER .
endptom)



(ptomod ALP-TEST is
 protecting ALP .
 protecting STRING .

 subsort String < Oid .

 op plane : Oid PlaneType Time Time Time Cost Cost Cost -> Configuration .
 op runway : Oid -> Configuration .
 op init1 : -> GlobalSystem .

 vars E T L : Time .
 vars C1 C2 C3 : Cost .
 var PT : PlaneType .
 var O : Oid .

 eq plane(O, PT, E, T, L, C1, C2, C3) =
          < O : Plane | type : PT, landAt : noOid, landingTime : T,
                        clock : 0, early : C1, late  : C2, latePenalty : C3,
                        earliest : E, target : T, latest : L, rate : 0 > .

 eq runway(O) =  < O : Runway | landed : nil, type : None, prevStart : 0 > .

 --- 3 planes and one runway
 eq init1 = {plane("p1", small, 5, 9, 12, 15, 2,3)
             plane("p2", small, 5, 5, 10, 25, 5, 10)
             plane("p3", medium, 5, 7, 15, 20, 3, 5)
             runway("rw1")} .
endptom)



(priced find earliest init1 =>* {< "rw1" : Runway | >} with no cost limit.)

Result: {< "rw1" : Runway | landed :(("p1",5)("p2",6)("p3",7)),prevStart : 7,
                            type : big >} in time 7 with cost 30



(find cheapest init1 =>* {< "rw1" : Runway | >} with no time limit .)

Solution
ATTRIBUTES_OF_"rw1":AttributeSet -->
 landed :(("p2",5)("p3",7)("p1",9)),
 prevStart : 9,type : small ;
CLASS_OF_"rw1":Runway --> Runway ;
TIME_ELAPSED:Time --> 9 ; TOTAL_COST_INCURRED:Cost --> 0
```

A.3.2 ETGS

```
(omod OID-SET is
  protecting DEF-OID .
  protecting SET{Oid} * (sort Set{Oid} to OidSet,
                         sort NeSet{Oid} to NeOidSet) .

  vars OS OS' : OidSet .

  op _subset_ : OidSet OidSet -> Bool .
  eq OS subset (OS, OS') = true .
  eq OS subset OS' = false [owise] .
endom)
```

```
(omod MESSAGES is
  sort MsgType .

  op bcast : -> MsgType [ctor] .

  msg msg_from_to_ : MsgType Oid Oid -> Msg .
endom)

(omod MESSAGES-MULT1 is
  protecting MESSAGES .
  protecting OID-SET .

  op multimsg_from_to_ : MsgType Oid OidSet -> Configuration .

  var MT : MsgType .
  vars O O' : Oid .
  var OS : OidSet .

  --- Make a message for every Oid in the set
  eq multimsg MT from O to empty = none .
  eq multimsg MT from O to (O', OS) =
     (msg MT from O to O') (multimsg MT from O to OS) .

endom)

(tomod OID-TIME is
  sort OidTimePair .
  op _`,_ : Oid Time -> OidTimePair [ctor] .
endtom)

(view OidTime from TRIV to OID-TIME is
  sort Elt to OidTimePair .
endv)



(tomod OID-TIME-SET is
  protecting SET{OidTime} * (sort Set{OidTime} to OidTimeSet,
                             sort NeSet{OidTime} to NeOidTimeSet) .
endtom)

(ptomod ETGS is
  protecting OID-TIME-SET .
  protecting MESSAGES-MULT1 .
  protecting NAT-COST-DOMAIN .
  protecting NAT-TIME-DOMAIN-WITH-INF .



  sort TaskState .

  ops unprocessed processing done : -> TaskState [ctor] .
  class ProcDevice | currentTask : DefOid,
                     activeRate  : Cost,
                     idleRate    : Cost,
                     timer : TimeInf .

  class Task | dependsOn : OidSet,
               procTimes : OidTimeSet,
               bcastTime : Time,
               status    : TaskState .

  class Processor | knownRes : OidSet .
  subclass Processor < ProcDevice .

  class Bus | connectedTo : OidSet .
  subclass Bus < ProcDevice .

  var R : Time .
  var TI : TimeInf .
  var SSt : SystemState .
  vars T P B : Oid .
  var C : Cost .
  var OS OS' : OidSet .

  crl [startTaskOnProcessor] :
      < T:Oid : Task | status : unprocessed,
                       procTimes : ((P:Oid, R:Time),
                                    OTS:OidTimeSet),
                       dependsOn : OS:OidSet >
      < P : Processor | timer : INF, knownRes : OS':OidSet >
      =>
      < T : Task | status : processing >
      < P : Processor | currentTask : T, timer : R >
  if (OS:OidSet subset OS':OidSet) .

  rl [finishTaskOnProcessor] :
     < T : Task | status : processing >
     < P : Processor | currentTask : T, timer : 0, knownRes : OS >
     =>
     < T : Task | status : done >
     < P : Processor | currentTask : noOid, knownRes : (OS:OidSet, T),
                       timer : INF > .

  rl [startBroadcast] :
     < T : Task | status : done, bcastTime : R >
```

```
    < B : Bus | timer : INF >
    =>
    < B : Bus | currentTask : T,
                timer : R > .

  rl [finishBroadcast] :
    < B : Bus | currentTask : T, timer : 0,
                connectedTo : OS >
    =>
    < B : Bus | currentTask : noOid, timer : INF >
    multimsg bcast from T to OS .

  rl [receiveBroadcast] :
    (msg bcast from T to P)
    < P : Processor | knownRes : OS >
    =>
    < P : Processor | knownRes : (OS, T) > .


  crl [tick] :
      {SSt} => {delta(SSt, R)} in time R with cost (rate(SSt) * R)
  if R <= mte(SSt) [nonexec] .


  eq mte(M:Msg) = 0 .
  eq mte(< T : Task | >) = INF .
  eq mte(< P : ProcDevice | timer : TI >) = TI .

  eq delta(< T : Task | >, R)
          =
          < T : Task | > .
  --- Processors and buses are affected by time.
  --- A processor or bus' timer counts down during the elapse of time.
  eq delta(< P : ProcDevice | timer : TI >, R)
     =
     < P : ProcDevice | timer : (TI monus R) > .

  eq rate(< P : ProcDevice | idleRate : C, timer : INF >) = C .
  eq rate(< P : ProcDevice | activeRate : C, timer : R >) = C .
  eq rate(< T : Task | >) = 0 .
endptom)


(ptomod ETGS-TEST is
 protecting NAT .
 protecting ETGS .
 protecting STRING .

 subsort String < Oid .

 op init : -> GlobalSystem .

 eq init = {< "t1"   : Task | dependsOn : empty, procTimes : ("p1", 1),
                              bcastTime : 7, status : unprocessed >
            < "t2"   : Task | dependsOn : empty, procTimes : ("p2", 2),
                              bcastTime : 5, status : unprocessed >
            < "t3"   : Task | dependsOn : ("t1", "t2"),
                              procTimes : (("p1", 5),  ("p2", 4)),
                              bcastTime : 0, status : unprocessed >
            < "p1" : Processor | currentTask : noOid, timer : INF,
                                 knownRes : empty, activeRate : 5,
                                 idleRate : 1 >
            < "p2" : Processor | currentTask : noOid, timer : INF,
                                 knownRes : empty, activeRate : 4,
                                 idleRate : 1 >
            < "bus"    : Bus | currentTask : noOid, timer : INF,
                               connectedTo : ("p1", "p2"),
                               activeRate : 11, idleRate    : 1 >} .
endptom)


(find cheapest init =>* {S:SystemState < "t3" : Task | status : done >} in time <= 12 .)


Solution
CLASS_OF_"t3":Task --> Task ;
    REMAINING_ATTRIBUTES_OF_"t3":AttributeSet -->
        bcastTime : 0, dependsOn :("t1", "t2"), procTimes :("p1",5, "p2",4);
S:SystemState -->
    < "bus" : Bus | activeRate : 11, connectedTo :("p1","p2"), currentTask : noOid,
                    idleRate : 1, timer : INF >
    < "p1" : Processor | activeRate : 5,currentTask : noOid, idleRate : 1,
                         knownRes :("t1", "t2", "t3"), timer : INF >
    < "p2" : Processor | activeRate : 4, currentTask : noOid, idleRate : 1,
                         knownRes : "t2", timer : INF >
    < "t1" : Task | bcastTime : 7, dependsOn : empty, procTimes : "p1",1, status : done > ;
    TIME_ELAPSED:Time --> 12 ; TOTAL_COST_INCURRED:Cost --> 116
```

A.3.3 SPR

```
fmod GRAPH is
```

```
  sorts Node Edge Graph .
  subsort Edge < Graph .

  op _->_ : Node Node -> Edge [ctor prec 40].
  op nil : -> Graph [ctor].
  op _;_ : Graph Graph -> Graph [ctor assoc comm id: nil].

  var M N O : Node .
  var G  : Graph .

  op reachable : Graph Node Node -> Bool .
  eq reachable(M -> N ; G, M, N) = true .
  ceq reachable(M -> O ; G, M, N) = true
  if reachable(G, O, N) = true .
  eq reachable(G, M, N) = false [owise].
endfm

(omod OID-LIST is
  protecting DEF-OID .
  protecting LIST{Oid} * (sort List{Oid} to OidList,
                          sort NeList{Oid} to NeOidList) .
endom)

(fmod SUBWAY-LAYOUT is
  protecting GRAPH .
  protecting OID-LIST .

  subsort Oid < Node .

  var G : Graph .
  vars St St' : Oid .
  var OL : OidList .
  op _reachable'from_in_ : Oid OidList Graph -> Bool .
  --- It is clearly sufficient to check the first entry of the route.
  --- All the entries after the first are reachable from the first.
  --- Thus by transitivity any stations reachable from any stations
  --- after the first are reachable from the first.
  eq St reachable from St' OL in G =
     reachable(G, St':Oid, St:Oid) .
  eq St reachable from nil in G  = false .
endfm)

(omod PASSENGER is
  sort PassengerGroup .
  op _going'to_ : NzNat Oid -> PassengerGroup [ctor] .
endom)

(view PassengerGroup from TRIV to PASSENGER is
  sort Elt to PassengerGroup .
endv)

(omod PASSENGER-SET is
  protecting SET{PassengerGroup} *
                       (sort Set{PassengerGroup} to PassengerSet,
                        sort NeSet{PassengerGroup} to NePassengerSet) .

  vars PS PS' : PassengerSet .
  vars Mz Nz : NzNat .
  vars O O' : Oid .

  op sumpas : PassengerSet -> Nat .
  eq sumpas(((Mz going to O), (Nz going to O'), PS)) =
             Mz + Nz + sumpas(PS) .
  eq sumpas(Mz going to O) = Mz .
  eq sumpas(empty) = 0 .

  eq Nz going to O, Mz going to O = (Nz + Mz) going to O .

  op remove_from_ : PassengerSet PassengerSet -> PassengerSet .
  eq remove empty from PS = PS .
  eq (remove ((Mz going to O),PS) from ((Nz going to O), PS')) =
     if Nz == Mz then
      empty
     else
      (sd(Nz, Mz) going to O)
     fi
      ,(remove PS from PS') .
endom)

(tomod STATION-SCHEDULE-ENTRY is
  protecting PASSENGER-SET .
  sort StationScheduleEntry .
  op ps : Time PassengerSet -> StationScheduleEntry [ctor] .
endtom)

(view StationScheduleEntry from TRIV to STATION-SCHEDULE-ENTRY is
  sort Elt to StationScheduleEntry .
endv)

(tomod STATION-SCHEDULE is
  protecting LIST{StationScheduleEntry} *
              (sort List{StationScheduleEntry} to StationSchedule,
               sort NeList{StationScheduleEntry} to NeStationSchedule) .
endtom)
```

```
(fmod SUBWAY-FUNCTIONS is
  protecting NAT .
  protecting OID-LIST .
  protecting SUBWAY-LAYOUT .
  protecting STATION-SCHEDULE .

  vars M N : Nat .
  vars Mz Nz : NzNat .
  vars O O' : Oid .
  var OL : OidList .
  vars PS PS' : PassengerSet .
  var G : Graph .

  op dropoff : Oid OidList PassengerSet Graph -> PassengerSet .
  eq dropoff(O, OL, ((Nz going to O), PS), G) =
     (Nz going to O), dropoff(O, OL, PS, G) .
  ceq dropoff(O, OL, ((Nz going to O'), PS), G) =
      if not (O' reachable from OL in G or (occurs(O',OL))) then
       (Nz going to O')
      else
       empty
      fi
      , dropoff(O, OL, PS, G)
  if O =/= O' .
  eq dropoff(O, OL, empty, G) = empty .

  op pickup : Nat OidList PassengerSet Graph -> PassengerSet .
  eq pickup(Mz, OL, ((Nz going to O), PS), G) =
     if (O reachable from OL in G) or (occurs(O,OL)) then
      if Mz <= Nz then
       (Mz going to O)
      else
       (Nz going to O), pickup(sd(Mz, Nz), OL, PS, G)
      fi
     else
      pickup(Mz, OL, PS, G)
     fi .
  eq pickup(Mz, OL, empty, G) = empty .
  eq pickup(0, OL, PS, G) = empty .
endfm)

(ptomod SUBWAY is
  protecting SUBWAY-FUNCTIONS .
  protecting OID-SET .
  protecting OID-LIST .
  protecting NAT-COST-DOMAIN .
  protecting NAT-TIME-DOMAIN-WITH-INF .

  sorts StationAction TrainStatus .

  ops attached detached boarded arrived : -> StationAction [ctor] .
  op _at`station_ : StationAction Oid -> TrainStatus .
  op traveling`to_ : Oid -> TrainStatus .

  class Train | cars : NzNat, route : OidList, visited : OidList,
                status : TrainStatus, passengers : PassengerSet,
                activeLO : Graph, passiveLO : Graph,
                timer : TimeInf .

   class Station | trains : OidSet,
                   passengerSchedule : StationSchedule,
                   passengers : PassengerSet,
                   clock : Time .

  class SeStation | cars : Nat .
  subclass SeStation < Station .

  op traveltime : Oid Oid -> Time [comm] .
  op carcap : -> Nat .
  ops attachcost detachcost movingcost : -> Cost .
  op cars : Nat -> Nat .
  op carcost : -> Cost .

  ops allowAttach allowDetach : -> Bool .

  vars St St' Tr : Oid .
  vars Ro Vi : OidList .
  var OS : OidSet .
  vars R C1 : Time .
  var Ti : TimeInf .
  vars M N : Nat .
  vars Mz Nz Ca : NzNat .
  vars PS PS' PS'' ON OFF STAY : PassengerSet .
  var SA : StationAction .
  var PSc : StationSchedule .
  vars AL PL : Graph .
  var PrS : PricedSystem .
  var SSt : SystemState .

  rl [arriveAtStation] :
     < Tr : Train | status : traveling to St, timer : 0,
                    visited : Vi >
     < St : Station | trains : OS >
     =>
     < Tr : Train | status : arrived at station St,
```

```
                  visited : St Vi >
     < St : Station | trains : (OS, Tr) > .

  crl [attachCart] :
      < Tr : Train | status : SA at station St, cars : Ca >
      < St : SeStation | cars : Nz >
      =>
      < Tr : Train | status : attached at station St,
                     cars : (Ca + 1) >
      < St : SeStation | cars : (Nz monus 1) >
      with cost attachcost
  if allowAttach and
     (SA == arrived or SA == attached) .


  crl [LoadUnloadPassengers] :
      < Tr : Train | status : SA at station St, cars : Ca,
                     passengers : PS, route : Ro, activeLO : AL >
      < St : Station | passengers : PS' >
      =>
      < Tr : Train | status : boarded at station St,
                     passengers : (STAY, ON) >
      < St : Station | passengers : ((remove ON from PS'), OFF) >
  if ((SA == arrived) or (SA == attached) or (SA == detached))
     /\ (OFF := dropoff(St, Ro, PS, AL)) /\ STAY := (remove OFF from PS)
     /\ (ON := pickup(((Ca * carcap) monus sumpas(STAY)), Ro, PS', AL)) .


  crl [detachCart] :
      < Tr : Train | status : SA at station St, cars : Ca,
                     passengers : PS >
      < St : SeStation | cars : N >
      =>
      < Tr : Train | status : detached at station St,
                     cars : (Ca monus 1) >
      < St : SeStation | cars : (N + 1) >
      with cost detachcost
  if allowDetach and Ca >= 2 and (cars(sumpas(PS)) < Ca)
     and (SA == boarded or SA == detached) .

  crl [trainDepart] :
     < Tr : Train | status : SA at station St, route : St' Ro >
     < St : Station | trains : (OS, Tr) >
     =>
     < Tr : Train | status : traveling to St',
                    timer : traveltime(St,St'), route : Ro >
     < St : Station | trains : OS >
  if SA == boarded or SA == detached .

  crl [tick] :
      {SSt} => {delta(SSt, R)} in time R with cost (rate(SSt) * R)
  if R <= mte(SSt) [nonexec] .

  eq delta(< St : Station | clock : Cl >, R)
           =
           < St : Station | clock : Cl + R > .

  eq mte(< St : Station | passengerSchedule : nil >) = INF .
  eq mte(< St : Station | clock : R, passengerSchedule : ps(Cl, PS) PSc >) = Cl monus R .
  eq rate(< St : Station | >) = 0 .

  --- Trains are affected by the elapse of time.
  --- The rate dependant on the number of cars
  eq mte(< Tr : Train | timer : Ti >) = Ti .
  eq delta(< Tr : Train | timer : Ti >, R)
     =
     < Tr : Train | timer : (Ti monus R) > .
  eq rate(< Tr : Train | cars : Ca >) = Ca * movingcost .

  eq < St : Station | clock : Cl, passengers : PS,
                      passengerSchedule : ps(Cl,PS') PSc >
     =
     < St : Station | clock : Cl, passengers : (PS, PS'),
                      passengerSchedule : PSc > .

  eq cars(N) = N quo carcap +
               if N rem carcap =/= 0 then 1 else 0 fi .

  eq < Tr : Train | status : SA at station St, visited : St Vi,
                    route : nil, activeLO : AL, passiveLO : PL > =
     < Tr : Train | visited : St, route : Vi, activeLO : PL,
                    passiveLO : AL > .

  eq < St : Station | passengers : ((Nz going to St), PS)  >
     =
     < St : Station | passengers : PS > .
endptom)

(ptomod SUBWAY-TEST is
  protecting SUBWAY .
  protecting STRING .

  subsort String < Oid .
  ops t1 t2 : -> Configuration .
```

```
  ops st1 st2 st3 st4 st5 st6 : -> Configuration .
  ops s1 s2 s3 s4 s5 s6 : -> Oid .
  ops init1 : -> GlobalSystem .

  eq s1 = "st1" .
  eq s2 = "st2" .
  eq s3 = "st3" .
  eq s4 = "st4" .
  eq s5 = "st5" .
  eq s6 = "st6" .

  eq traveltime("st1", "st2") = 2 .
  eq traveltime("st2", "st3") = 3 .
  eq traveltime("st3", "st4") = 3 .
  eq traveltime("st3", "st5") = 1 .
  eq traveltime("st3", "st6") = 4 .

  ops route1a route1b route2a route2b l1uc l2uc : -> Graph .
  eq route1a = s1 -> s2 ; s2 -> s3 ; s3 -> s4 .
  eq route1b = s4 -> s3 ; s3 -> s2 ; s2 -> s1 .
  eq l1uc = s3 -> s5 ; s3 -> s6 .
  eq route2a = s5 -> s3 ; s3 -> s6 .
  eq route2b = s6 -> s3 ; s3 -> s5 .
  eq l2uc    = s3 -> s4 ; s3 -> s2 ; s2 -> s1 .

  eq carcap = 10 .    --- passenger capacity of each car
  eq attachcost = 5 . --- cost of attaching new car
  eq detachcost = 4 . --- cost of detaching a car
  eq movingcost = 3 . --- cost of movement per car

  eq allowAttach = false . --- disallow attaching new cars
  eq allowDetach = false . --- disallow detaching cars

  eq t1 = < "t1" : Train | cars : 3,
                           route : s2 s3 s4,
                           visited : nil,
                           status : traveling to s1,
                           timer : 0,
                           passengers : empty,
                           activeLO  : route1a ; l1uc,
                           passiveLO : route1b ; l1uc > .


 eq t2 = < "t2" : Train | cars : 3,
                          route : s3 s6,
                          visited : nil,
                          status : traveling to s5,
                          timer : 0,
                          passengers : empty,
                          activeLO   : route2a ; l2uc,
                          passiveLO  : route2b ; l2uc > .


  eq st1 = < s1 : Station | trains         : empty,
                            passengers     : empty,
                            passengerSchedule : nil,
                            clock : 0 > .

  eq st2 = < s2 : Station | trains         : empty,
                            passengers     : empty,
                            passengerSchedule :
                                    ps(2,(11 going to s1,
                                           1 going to s3,
                                           1 going to s5,
                                           2 going to s6))
                                    ps(7,(2 going to s1,
                                          5 going to s3,
                                          7 going to s4,
                                          4 going to s5)),
                            clock : 0 > .

  eq st3 = < s3 : SeStation | trains         : empty,
                              cars           : 1,
                              passengers     : empty,
                              passengerSchedule : nil,
                              clock : 0 > .

  eq st4 = < s4 : Station | trains         : empty,
                            passengers     : empty,
                            passengerSchedule : nil,
                            clock : 0 > .

  eq st5 = < s5 : Station | trains         : empty,
                            passengers     : empty,
                            passengerSchedule : nil,
                            clock : 0 > .

  eq st6 = < s6 : Station | trains         : empty,
                            passengers     : 1 going to s1,
                            passengerSchedule : nil,
                            clock : 0 > .

  eq init1 = t1 t2 st1 st2 st3 st4 st5 st6 .
endptom)
```

```
(priced find earliest init1 =>*
        {< "t1" : Train | passengers : empty >
        < "t2" : Train | passengers : empty >
        < "st1" : Station | passengers : empty >
        < "st2" : Station | passengers : empty >
        < "st3" : Station | passengers : empty >
        < "st4" : Station | passengers : empty >
        < "st5" : Station | passengers : empty >
       < "st6" : Station | passengers : empty >} with not cost limit.)

Result:
{...} in time 28 with cost 504


(find cheapest init1 =>*
       {< "t1" : Train | passengers : empty >
        < "t2" : Train | passengers : empty >
        < "st1" : Station | passengers : empty >
        < "st2" : Station | passengers : empty >
        < "st3" : Station | passengers : empty >
        < "st4" : Station | passengers : empty >
        < "st5" : Station | passengers : empty >
        < "st6" : Station | passengers : empty >} with no time limit .)

Solution
....
TIME_ELAPSED:Time --> 28 ; TOTAL_COST_INCURRED:Cost --> 504


(ptsearch [1] init1 =>*
       {S:SystemState
        < O:Oid : Station | passengers : 1 going to "st4" >}
        such that
         O:Oid =/= "st2" and O:Oid =/= "st4"
         in time <= 30 with no cost limit.)

No solution
```

A.4 Example Listings: UPPAAL CORA Specifications

This section lists three of the UPPAAL CORA specifications used for testing in Chapter 8. One specification for two, three, and four processors is listed.

A.4.1 Two Processors, Ten Tasks, and a Bus

This first specification, models a system with two processors, ten tasks, and a bus. Önly the deadlines and number of tasks need to be changed for it to identical to all the other specifications with the same amount of processors.

```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN' '
http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'><nta><declaration>//constants used to identify processors in the various arrays
const int bus = 0;
const int p1 = 1;
const int p2 = 2;

const int t1 = 0;
const int t2 = 1;
const int t3 = 2;
const int t4 = 3;
const int t5 = 4;
const int t6 = 5;
const int t7 = 6;
const int t8 = 7;
const int t9 = 8;
const int t10 = 9;

// Place global declarations here.
const int tasks = 10;
const int procs = 2;

//what processors or buses are active
bool act[procs + 1];

//remaining running/transmission time
int d[procs + 1];

//array to keep track of what results have been received
bool res[procs + 1][tasks];

chan pbus[procs + 1];
```

```
//array of active rates
const int pi[procs + 1] = {11,5,4};
//array of idle rates
const int tau[procs + 1] = {1,1,1};


//broadcast times (kappa), run times on p1, run times on p2
const int na = -1;
const int delta[tasks][procs +1] = {{ 7,  1, na}, //t1
                    { 5, na,  2}, //t2
                    { 6,  5,  4}, //t3
                    { 5,  2,  4}, //t4
                    { 6,  3,  2}, //t5
                    { 4,  2,  5}, //t6
                    { 2,  2,  1}, //t7
                    { 3,  4,  3}, //t8
                    { 4,  2,  4}, //t9
                    { 3,  3,  1}}; //t10

const int none = -1;
const int pre[tasks][2] =  {{none, none}, //t1
                {none, none}, //t2
                {  t1,   t2}, //t3
                {none,   t2}, //t4
                {  t3,   t4}, //t5
                {  t4,   t5}, //t6
                {  t6, none}, //t7
                {  t7, none}, //t8
                {none,   t2}, //t9
                {  t9,   t8}}; //t10

clock systemClock;
const int dl = 24;</declaration><template><name>processor</name><parameter>
const int p, const int tau, const int pi</parameter><declaration>//internal clock for this processor
clock c;</declaration><location id="id0" x="-64" y="-32"><name x="-74" y="-62">active</name>
<label kind="invariant" x="-96" y="0">c &lt;= d[p] &amp;&amp;
cost' == pi</label></location><location id="id1" x="-64" y="-136"><name x="-74" y="-166">idle
</name><label kind="invariant" x="-96" y="-192">cost' == tau</label></location><init ref="id1"/>
<transition><source ref="id1"/><target ref="id0"/><label kind="synchronisation" x="0" y="-112">pbus[p]?</label>
<label kind="assignment" x="8" y="-80">c := 0,act[p] := true</label><nail x="-24" y="-120"/><nail x="-16" y="-48"/>
</transition><transition><source ref="id0"/><target ref="id1"/><label kind="guard" x="-176" y="-120">c == d[p]
</label><label kind="synchronisation" x="-168" y="-104">pbus[p]!
</label><label kind="assignment" x="-232" y="-64">act[p] := false</label><nail x="-112" y="-56"/><nail x="-112" y="-112"/>
</transition></template><template><name>task2
</name><parameter>const int task, const int pre1, const int pre2, const int delta1, const int delta2, const int kappa
</parameter><location id="id2" x="-232" y="24"></location><location id="id3" x="-240" y="-128"><name x="-192" y="-144">broadcasting
</name></location><location id="id4" x="-248" y="-384"><name x="-248" y="-424">done
</name><label kind="invariant" x="-312" y="-448">systemClock &lt;= dl
</label></location><location id="id5" x="-128" y="-384"><name x="-88" y="-392">processing_2</name>
</location><location id="id6" x="-368" y="-384"><name x="-480" y="-392">processing_1</name>
</location><location id="id7" x="-256" y="-560"><name x="-224" y="-600">unprocessed</name>
</location><init ref="id7"/><transition><source ref="id3"/><target ref="id2"/><label kind="synchronisation" x="-192" y="-72">pbus[bus]?
</label><label kind="assignment" x="-192" y="-48">res[p1][task] := true, res[p2][task] := true
</label></transition><transition><source ref="id4"/><target ref="id3"/><label kind="guard" x="-208" y="-256">not act[bus]
</label><label kind="synchronisation" x="-208" y="-240">pbus[bus]!
</label><label kind="assignment" x="-208" y="-216">d[bus] := kappa</label></transition><transition><source ref="id6"/><target ref="id4"/>
<label kind="synchronisation" x="-360" y="-336">pbus[p1]?</label><label kind="assignment" x="-384" y="-360">res[p1][task] := true
</label></transition><transition><source ref="id5"/><target ref="id4"/><label kind="synchronisation" x="-216" y="-336">pbus[p2]?
</label><label kind="assignment" x="-216" y="-360">res[p2][task] := true
</label></transition><transition><source ref="id7"/><target ref="id5"/><label kind="guard" x="-88" y="-648">delta2 != na
and
(pre1 == none or res[p2][pre1] == 1)
and
(pre2 == none or res[p2][pre2] == 1)
and
not act[p2]</label><label kind="synchronisation" x="-160" y="-520">pbus[p2]!
</label><label kind="assignment" x="-152" y="-488">d[p2] := delta2
</label></transition><transition><source ref="id7"/><target ref="id6"/><label kind="guard" x="-560" y="-648">delta1 != na
and
(pre1 == none or res[p1][pre1] == 1)
and
(pre2 == none or res[p1][pre2] == 1)
and
not act[p1]</label><label kind="synchronisation" x="-416" y="-504">pbus[p1]!
</label><label kind="assignment" x="-448" y="-480">d[p1] := delta1</label>
</transition></template><system>// Place template instantiations here.
task_t1 = task2(t1, pre[t1][0], pre[t1][1], delta[t1][p1], delta[t1][p2], delta[t1][bus]);
task_t2 = task2(t2, pre[t2][0], pre[t2][1], delta[t2][p1], delta[t2][p2], delta[t2][bus]);
task_t3 = task2(t3, pre[t3][0], pre[t3][1], delta[t3][p1], delta[t3][p2], delta[t3][bus]);
task_t4 = task2(t4, pre[t4][0], pre[t4][1], delta[t4][p1], delta[t4][p2], delta[t4][bus]);
task_t5 = task2(t5, pre[t5][0], pre[t5][1], delta[t5][p1], delta[t5][p2], delta[t5][bus]);
task_t6 = task2(t6, pre[t6][0], pre[t6][1], delta[t6][p1], delta[t6][p2], delta[t6][bus]);
task_t7 = task2(t7, pre[t7][0], pre[t7][1], delta[t7][p1], delta[t7][p2], delta[t7][bus]);
task_t8 = task2(t8, pre[t8][0], pre[t8][1], delta[t8][p1], delta[t8][p2], delta[t8][bus]);
task_t9 = task2(t9, pre[t9][0], pre[t9][1], delta[t9][p1], delta[t9][p2], delta[t9][bus]);
task_t10 = task2(t10, pre[t10][0], pre[t10][1], delta[t10][p1], delta[t10][p2], delta[t10][bus]);

proc_bus = processor(bus,tau[bus],pi[bus]);
proc_p1 = processor(p1,tau[p1],pi[p1]);
proc_p2 = processor(p2,tau[p2],pi[p2]);
// List one or more processes to be composed into a system.

system proc_bus, proc_p1, proc_p2, task_t1, task_t2, task_t3, task_t4, task_t5, task_t6, task_t7, task_t8, task_t9, task_t10;

</system></nta>
```

A.4.2 Three Processors, Ten Tasks, and a Bus

The second specification, models a system with three processors, ten tasks, and a bus. Önly the deadlines and number of tasks need to be changed for it to identical to all the other specifications with the same amount of processors.


```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN' '
http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'><nta><declaration>//constants used to identify processors in the various arrays
const int bus = 0;
const int p1 = 1;
const int p2 = 2;
const int p3 = 3;

const int t1 = 0;
const int t2 = 1;
const int t3 = 2;
const int t4 = 3;
const int t5 = 4;
const int t6 = 5;
const int t7 = 6;
const int t8 = 7;
const int t9 = 8;
const int t10 = 9;

// Place global declarations here.
const int tasks = 10;
const int procs = 3;

//what processors or buses are active
bool act[procs + 1];

//remaining running/transmission time
int d[procs + 1];

//array to keep track of what results have been received
bool res[procs + 1][tasks];

chan pbus[procs + 1];

//array of active rates
const int pi[procs + 1] = {11,5,4,4};
//array of idle rates
const int tau[procs + 1] = {1,1,1,1};

//array holding: {broadcast time, delta p1, delta p2, delta p4}
const int na = -1;
const int delta[tasks][procs +1] = {{7,  1, na, na}, //t1
                    {5, na,  2, na}, //t2
                    {6,  5,  4,  2}, //t3
                    {5,  2,  4, na}, //t4
                    {6,  3,  2,  2}, //t5
                    {4,  2,  3, na}, //t6
                    {2,  3,  1,  2}, //t7
                    {3,  4,  3,  4}, //t8
                    {4,  2,  4, na}, //t9
                    {3,  3,  1,  2}}; //t10

//array holding: {predecessor 1, predecessor 2}
const int none = -1;
const int pre[tasks][2] =   {{none, none}, //t1
                {none, none}, //t2
                {  t1,   t2}, //t3
                {none,   t2}, //t4
                {  t3,   t4}, //t5
                {  t4,   t5}, //t6
                {  t6, none}, //t7
                {  t7, none}, //t8
                {none,   t2}, //t9
                {  t9,   t8}}; //t10

clock systemClock;
const int dl = 24;

</declaration><template><name>processor</name><parameter> const int p, const int tau, const int pi
</parameter><declaration>//internal clock for this processor
clock c;</declaration><location id="id0" x="-64" y="-32"><name x="-74" y="-62">active
</name><label kind="invariant" x="-96" y="0">c &lt;= d[p] &amp;amp;&amp;
cost' == pi</label></location><location id="id1" x="-64" y="-136"><name x="-74" y="-166">idle
</name><label kind="invariant" x="-96" y="-192">cost' == tau</label></location><init ref="id1"/>
<transition><source ref="id1"/><target ref="id0"/><label kind="synchronisation" x="0" y="-112">pbus[p]?
</label><label kind="assignment" x="8" y="-80">c := 0,
act[p] := true</label><nail x="-24" y="-120"/><nail x="-16" y="-48"/></transition><transition><source ref="id0"/><target ref="id1"/>
<label kind="guard" x="-176" y="-120">c == d[p]</label><label kind="synchronisation" x="-168" y="-104">pbus[p]!
</label><label kind="assignment" x="-232" y="-64">act[p] := false</label><nail x="-112" y="-56"/><nail x="-112" y="-112"/>
</transition></template><template><name>task3</name>
<parameter>const int task, const int pre1, const int pre2, const int delta1, const int delta2, const int delta3, const int kappa
</parameter><location id="id2" x="-248" y="-376"><name x="-224" y="-384">processing_3</name>
</location><location id="id3" x="-232" y="64"></location><location id="id4" x="-232" y="-80"><name x="-184" y="-96">broadcasting</name>
</location><location id="id5" x="-240" y="-224"><name x="-240" y="-264">done</name>
<label kind="invariant" x="-384" y="-232">systemClock &lt;= dl</label></location><location id="id6" x="184" y="-432">
<name x="224" y="-440">processing_2</name></location><location id="id7" x="-624" y="-432"><name x="-736" y="-440">processing_1
```

```
</name></location><location id="id8" x="-256" y="-640"><name x="-224" y="-680">unprocessed</name>
</location><init ref="id8"/><transition><source ref="id2"/><target ref="id5"/>
<label kind="synchronisation" x="-328" y="-336">pbus[p3]?</label><label kind="assignment" x="-368" y="-352">res[p3][task] := true</label>
</transition><transition><source ref="id8"/><target ref="id2"/><label kind="guard" x="-232" y="-536">delta3 != na
and
(pre1 == none or res[p3][pre1])
and
(pre2 == none or res[p3][pre2])
and
not act[p3]</label><label kind="synchronisation" x="-328" y="-520">pbus[p3]!</label><label kind="assignment" x="-352" y="-504">d[p3] := delta3
</label></transition><transition><source ref="id4"/><target ref="id3"/><label kind="synchronisation" x="-216" y="-32">pbus[bus]?
</label><label kind="assignment" x="-216" y="-16">res[p1][task] := true,
res[p2][task] := true,
res[p3][task] := true</label></transition><transition><source ref="id5"/><target ref="id4"/>
<label kind="guard" x="-216" y="-208">not act[bus]</label><label kind="synchronisation" x="-216" y="-184">pbus[bus]!
</label><label kind="assignment" x="-216" y="-160">d[bus] := kappa</label></transition><transition><source ref="id7"/><target ref="id5"/>
<label kind="synchronisation" x="-536" y="-336">pbus[p1]?</label><label kind="assignment" x="-592" y="-360">res[p1][task] := true</label>
</transition><transition><source ref="id6"/><target ref="id5"/><label kind="synchronisation" x="-48" y="-320">pbus[p2]?
</label><label kind="assignment" x="-16" y="-336">res[p2][task] := true</label></transition><transition><source ref="id8"/><target ref="id6"/>
<label kind="guard" x="56" y="-680">delta2 != na
and
(pre1 == none or res[p2][pre1])
and
(pre2 == none or res[p2][pre2])
and
not act[p2]</label><label kind="synchronisation" x="56" y="-576">pbus[p2]!</label><label kind="assignment" x="56" y="-560">d[p2] := delta2
</label></transition><transition><source ref="id8"/><target ref="id7"/><label kind="guard" x="-560" y="-680">delta1 != na
and
(pre1 == none or res[p1][pre1])
and
(pre2 == none or res[p1][pre2])
and
not act[p1]</label><label kind="synchronisation" x="-560" y="-576">pbus[p1]!
</label><label kind="assignment" x="-560" y="-560">d[p1] := delta1</label>
</transition></template><system>// Place template instantiations here.
task_t1 = task3(t1, pre[t1][0], pre[t1][1], delta[t1][p1], delta[t1][p2], delta[t1][p3], delta[t1][bus]);
task_t2 = task3(t2, pre[t2][0], pre[t2][1], delta[t2][p1], delta[t2][p2], delta[t2][p3], delta[t2][bus]);
task_t3 = task3(t3, pre[t3][0], pre[t3][1], delta[t3][p1], delta[t3][p2], delta[t3][p3], delta[t3][bus]);
task_t4 = task3(t4, pre[t4][0], pre[t4][1], delta[t4][p1], delta[t4][p2], delta[t4][p3], delta[t4][bus]);
task_t5 = task3(t5, pre[t5][0], pre[t5][1], delta[t5][p1], delta[t5][p2], delta[t5][p3], delta[t5][bus]);
task_t6 = task3(t6, pre[t6][0], pre[t6][1], delta[t6][p1], delta[t6][p2], delta[t6][p3], delta[t6][bus]);
task_t7 = task3(t7, pre[t7][0], pre[t7][1], delta[t7][p1], delta[t7][p2], delta[t7][p3], delta[t7][bus]);
task_t8 = task3(t8, pre[t8][0], pre[t8][1], delta[t8][p1], delta[t8][p2], delta[t8][p3], delta[t8][bus]);
task_t9 = task3(t9, pre[t9][0], pre[t9][1], delta[t9][p1], delta[t9][p2], delta[t9][p3], delta[t9][bus]);
task_t10 = task3(t10, pre[t10][0], pre[t10][1], delta[t10][p1], delta[t10][p2], delta[t10][p3], delta[t10][bus]);

proc_bus = processor(bus,tau[bus],pi[bus]);
proc_p1 = processor(p1,tau[p1],pi[p1]);
proc_p2 = processor(p2,tau[p2],pi[p2]);
proc_p3 = processor(p3,tau[p3],pi[p3]);
// List one or more processes to be composed into a system.

system proc_bus, proc_p1, proc_p2, proc_p3, task_t1, task_t2, task_t3, task_t4, task_t5, task_t6, task_t7, task_t8, task_t9, task_t10;

</system></nta>
```

A.4.3 Four Processors, Ten Tasks, and a Bus

The third specification, models a system with four processors, ten tasks, and a bus. Önly the deadlines and number of tasks need to be changed for it to identical to all the other specifications with the same amount of processors.

```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN' '
http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'><nta><declaration>//constants used to identify processors in the various arrays
const int bus = 0;
const int p1 = 1;
const int p2 = 2;
const int p3 = 3;
const int p4 = 4;

const int t1 = 0;
const int t2 = 1;
const int t3 = 2;
const int t4 = 3;
const int t5 = 4;
const int t6 = 5;
const int t7 = 6;
const int t8 = 7;
const int t9 = 8;
const int t10 = 9;

// Place global declarations here.
const int tasks = 10;
const int procs = 4;

//what processors or buses are active
bool act[procs + 1];
```

```
//remaining running/transmission time
int d[procs + 1];

//array to keep track of what results have been received
bool res[procs + 1][tasks];

chan pbus[procs + 1];

//array of active rates
const int pi[procs + 1] = {11,5,4,4,5};
//array of idle rates
const int tau[procs + 1] = {1,1,1,4,5};


//broadcast times (kappa), run times on p1, run times on p2
const int na = -1;
//array holding: {broadcast time, delta p1, delta p2, delta p4}
const int delta[tasks ][procs +1] = {{7,  1, na, na, na}, //t1
                    {5, na,  2, na, na}, //t2
                            {6,  5,  4,  2, na}, //t3
                    {5,  2,  4, na, na}, //t4
                    {6,  3,  2,  2, na}, //t5
                    {4,  2,  3, na, na}, //t6
                    {2,  3,  1,  2, na}, //t7
                    {3,  4,  3,  4, na}, //t8
                    {4,  2,  4, na, na}, //t9
                    {3,  3,  1,  2, na}}; //t10


const int none = -1;
//array holding: {predecessor 1, predecessor 2}
const int pre[tasks][2] =   {{none, none}, //t1
                {none, none}, //t2
                {  t1,   t2}, //t3
                {none,   t2}, //t4
                {  t3,   t4}, //t5
                {  t4,   t5}, //t6
                {  t6, none}, //t7
                {  t7, none}, //t8
                {none,   t2}, //t9
                {  t9,   t8}}; //t10

clock systemClock;
const int dl = 24;</declaration><template><name>processor</name><parameter> const int p, const int tau, const int pi
</parameter><declaration>//internal clock for this processor
clock c;</declaration><location id="id0" x="-64" y="-32"><name x="-74" y="-62">active
</name><label kind="invariant" x="-96" y="0">c &lt;= d[p] &amp;amp;&amp;
cost' == pi</label></location><location id="id1" x="-64" y="-136"><name x="-74" y="-166">idle
</name><label kind="invariant" x="-96" y="-192">cost' == tau</label></location><init ref="id1"/>
<transition><source ref="id1"/><target ref="id0"/><label kind="synchronisation" x="0" y="-112">pbus[p]?
</label><label kind="assignment" x="8" y="-80">c := 0,
act[p] := true</label><nail x="-24" y="-120"/><nail x="-16" y="-48"/></transition><transition><source ref="id0"/>
<target ref="id1"/><label kind="guard" x="-176" y="-120">c == d[p]</label><label kind="synchronisation" x="-168" y="-104">pbus[p]!
</label><label kind="assignment" x="-232" y="-64">act[p] := false</label><nail x="-112" y="-56"/><nail x="-112" y="-112"/>
</transition></template><template><name>task4</name>
<parameter>const int task, const int pre1, const int pre2, const int delta1, const int delta2, const int delta3, const int delta4, const int kappa
</parameter><location id="id2" x="-168" y="-448"><name x="-152" y="-440">processing_4</name>
</location><location id="id3" x="-440" y="-440"><name x="-416" y="-448">processing_3</name>
</location><location id="id4" x="-232" y="64"></location><location id="id5" x="-232" y="-80"><name x="-184" y="-96">broadcasting
</name></location><location id="id6" x="-240" y="-224"><name x="-304" y="-224">done
</name><label kind="invariant" x="-512" y="-240">systemClock &lt;= dl</label>
</location><location id="id7" x="248" y="-640"><name x="288" y="-648">processing_2</name>
</location><location id="id8" x="-680" y="-632"><name x="-792" y="-640">processing_1</name>
</location><location id="id9" x="-256" y="-640"><name x="-224" y="-680">unprocessed</name>
</location><init ref="id9"/><transition><source ref="id2"/><target ref="id6"/>
<label kind="synchronisation" x="-168" y="-384">pbus[p4]?</label><label kind="assignment" x="-168" y="-400">res[p4][task] := true
</label></transition><transition><source ref="id9"/><target ref="id2"/><label kind="guard" x="-160" y="-640">delta4 != na
and
(pre1 == none or res[p4][pre1])
and
(pre2 == none or res[p4][pre2])
and
not act[p4]</label><label kind="synchronisation" x="-160" y="-528">pbus[p4]!
</label><label kind="assignment" x="-160" y="-504">d[p4] := delta4</label>
</transition><transition><source ref="id3"/><target ref="id6"/><label kind="synchronisation" x="-472" y="-384">pbus[p3]?
</label><label kind="assignment" x="-528" y="-408">res[p3][task] := true</label>
</transition><transition><source ref="id9"/><target ref="id3"/><label kind="guard" x="-576" y="-624">delta3 != na
and
(pre1 == none or res[p3][pre1])
and
(pre2 == none or res[p3][pre2])
and
not act[p3]</label><label kind="synchronisation" x="-512" y="-504">pbus[p3]!
</label><label kind="assignment" x="-528" y="-488">d[p3] := delta3</label>
</transition><transition><source ref="id5"/><target ref="id4"/><label kind="synchronisation" x="-216" y="-32">pbus[bus]?
</label><label kind="assignment" x="-216" y="-16">res[p1][task] := true,
res[p2][task] := true,
res[p3][task] := true,
res[p4][task] := true</label></transition><transition><source ref="id6"/><target ref="id5"/>
<label kind="guard" x="-216" y="-208">not act[bus]</label><label kind="synchronisation" x="-216" y="-184">pbus[bus]!
</label><label kind="assignment" x="-216" y="-160">d[bus] := kappa</label>
</transition><transition><source ref="id8"/><target ref="id6"/><label kind="synchronisation" x="-632" y="-336">pbus[p1]?
</label><label kind="assignment" x="-656" y="-376">res[p1][task] := true
</label><nail x="-568" y="-408"/><nail x="-520" y="-352"/><nail x="-472" y="-304"/>
</transition><transition><source ref="id7"/><target ref="id6"/><label kind="synchronisation" x="40" y="-320">pbus[p2]?
```

```
</label><label kind="assignment" x="32" y="-336">res[p2][task] := true</label><nail x="88" y="-376"/><nail x="-32" y="-296"/>
</transition><transition><source ref="id9"/><target ref="id7"/><label kind="guard" x="56" y="-792">delta2 != na
and
(pre1 == none or res[p2][pre1])
and
(pre2 == none or res[p2][pre2])
and
not act[p2]</label><label kind="synchronisation" x="64" y="-688">pbus[p2]!
</label><label kind="assignment" x="64" y="-672">d[p2] := delta2</label>
</transition><transition><source ref="id9"/><target ref="id8"/><label kind="guard" x="-560" y="-792">delta1 != na
and
(pre1 == none or res[p1][pre1])
and
(pre2 == none or res[p1][pre2])
and
not act[p1]</label><label kind="synchronisation" x="-560" y="-688">pbus[p1]!
</label><label kind="assignment" x="-568" y="-664">d[p1] := delta1</label></transition></template><system>




// Place template instantiations here.
task_t1 = task4(t1, pre[t1][0], pre[t1][1], delta[t1][p1], delta[t1][p2], delta[t1][p3], delta[t1][p4], delta[t1][bus]);
task_t2 = task4(t2, pre[t2][0], pre[t2][1], delta[t2][p1], delta[t2][p2], delta[t2][p3], delta[t2][p4], delta[t2][bus]);
task_t3 = task4(t3, pre[t3][0], pre[t3][1], delta[t3][p1], delta[t3][p2], delta[t3][p3], delta[t3][p4], delta[t3][bus]);
task_t4 = task4(t4, pre[t4][0], pre[t4][1], delta[t4][p1], delta[t4][p2], delta[t4][p3], delta[t4][p4], delta[t4][bus]);
task_t5 = task4(t5, pre[t5][0], pre[t5][1], delta[t5][p1], delta[t5][p2], delta[t5][p3], delta[t5][p4], delta[t5][bus]);
task_t6 = task4(t6, pre[t6][0], pre[t6][1], delta[t6][p1], delta[t6][p2], delta[t6][p3], delta[t6][p4], delta[t6][bus]);
task_t7 = task4(t7, pre[t7][0], pre[t7][1], delta[t7][p1], delta[t7][p2], delta[t7][p3], delta[t7][p4], delta[t7][bus]);
task_t8 = task4(t8, pre[t8][0], pre[t8][1], delta[t8][p1], delta[t8][p2], delta[t8][p3], delta[t8][p4], delta[t8][bus]);
task_t9 = task4(t9, pre[t9][0], pre[t9][1], delta[t9][p1], delta[t9][p2], delta[t9][p3], delta[t9][p4], delta[t9][bus]);
task_t10 = task4(t10, pre[t10][0], pre[t10][1], delta[t10][p1], delta[t10][p2], delta[t10][p3], delta[t10][p4], delta[t10][bus]);

proc_bus = processor(bus,tau[bus],pi[bus]);
proc_p1 = processor(p1,tau[p1],pi[p1]);
proc_p2 = processor(p2,tau[p2],pi[p2]);
proc_p3 = processor(p3,tau[p3],pi[p3]);
proc_p4 = processor(p4,tau[p4],pi[p4]);
// List one or more processes to be composed into a system.

system proc_bus, proc_p1, proc_p2, proc_p3, proc_p4, task_t1, task_t2, task_t3, task_t4, task_t5, task_t6, task_t7, task_t8, task_t9, task_t10;

</system></nta>
```

Appendix B

Priced-Timed Maude Implementation Listings

```
*** the sorts System and GlobalSystem may also be
*** needed to contain priced untimed systems
fmod UNTIMED-PRELUDE is
   sorts System GlobalSystem .
   op {_} : System -> GlobalSystem [format (g o g so)] .
endfm

*** Modified by PRICED-timed-maude
*** TIMED-OO-PRELUDDE split in two
mod OO-PRELUDE is
  including CONFIGURATION .
  sorts EmptyConfiguration NEConfiguration MsgConfiguration
        NEMsgConfiguration ObjectConfiguration NEObjectConfiguration .
  subsorts EmptyConfiguration < MsgConfiguration ObjectConfiguration
                                < Configuration .
  subsorts Msg < NEMsgConfiguration < MsgConfiguration NEConfiguration .
  subsorts Object < NEObjectConfiguration <
             ObjectConfiguration NEConfiguration .
  subsort NEConfiguration < Configuration .

  --- op none : -> EmptyConfiguration . --- crashes w/ none for Configuration
  op __ : EmptyConfiguration EmptyConfiguration -> EmptyConfiguration [ditto] .
  op __ : NEConfiguration NEConfiguration -> NEConfiguration [ditto] .
  op __ : MsgConfiguration MsgConfiguration -> MsgConfiguration [ditto] .
  op __ : NEMsgConfiguration NEMsgConfiguration -> NEMsgConfiguration [ditto] .
  op __ : ObjectConfiguration ObjectConfiguration ->
                                  ObjectConfiguration [ditto] .
  op __ : NEObjectConfiguration NEObjectConfiguration ->
                                  NEObjectConfiguration [ditto] .
endm

*** ***********************************************
*** ***********************************************
***
***   PRICED-TIMED MAUDE stuff
***
*** ***********************************************
*** ***********************************************
--- For now all cost related stuff is added here

--- method borrowed from Real-Time Maude
fmod COST is
 sorts Cost NzCost CostInf .

 subsort NzCost < Cost < CostInf .

 op infcost : -> CostInf [ctor] .
 --- operators that are used on nats and rats need to be renamed or
 --- they weill conflict with rtms operators
 op _pluss_ : Cost Cost -> Cost [assoc comm id:free prec 33 gather (E e)] .

 op free : -> Cost .

 op _cheaper than_ : Cost Cost -> Bool [prec 37] .
 op _cheaper than or eq_ : Cost Cost -> Bool [prec 37] .

 vars C C' : Cost .

 eq C cheaper than or eq C' = (C cheaper than C') or (C == C') .

 --- infcost is a special value indicating: no term matching our
 --- query has been found
 eq C cheaper than infcost = true .
```

```
 eq infcost cheaper than C = false .
endfm

fmod DIVIDE-COST is
 inc COST .
 op div2 : Cost -> Cost .

 eq div2(free) = free .
endfm

--- defines max and min for costs
--- not strictly a definition of linear
--- is this one needed?
--- maybe there's a way to separate out the overloading of
--- minimum, maximum, lt, gt, le, ge, pkus and monus from
--- real-time maude and share it, for instance in a
--- nat-rat-domain.maude or something
fmod LCOST is
  including COST .

  ops minCost maxCost : Cost Cost -> Cost [assoc comm] .

  vars C C' : Cost .
  ceq maxCost(C, C') = C if C' cheaper than or eq C .
  ceq minCost(C, C') = C' if C' cheaper than or eq C .
endfm

fmod ABSTRACT-COST is
 including LCOST .
 including DIVIDE-COST .
endfm

--- costs represented by the domain of natural numbers
--- same techniques as for Time in Real-Time Maude
fmod NAT-COST-DOMAIN is
 including ABSTRACT-COST .
 protecting NAT .

 subsort Nat < Cost .
 subsort NzNat < NzCost .

 vars N N' : Nat .

 eq free = 0 .
 eq N pluss N' = N + N' .
 eq N cheaper than N' = N < N' .

 eq div2(N) = N quo 2 .
endfm


--- the following defines the postive rational numbers as the cost
--- domain much like this is done for time in Real-Time Maude
fmod POSRAT-COST-DOMAIN is
  including ABSTRACT-COST .
  inc POSITIVE-RAT .

  subsort NNegRat < Cost .
  subsort PosRat < NzCost .

  vars R R' : NNegRat .

  eq free = 0 .
  eq R pluss R' = R + R' .

  eq R cheaper than R' = R < R' .

  eq div2(R) = R / 2 .
endfm

fmod PRICED-SYSTEM is
 including UNTIMED-PRELUDE .
 protecting COST .
 *** Do we need something like [_] to contain the PricedSystem?
 sorts SystemState PricedSystem .
 subsort SystemState PricedSystem < System .

 op _with cost_ : PricedSystem Cost
    -> PricedSystem [prec 95 gather (E e)] .
endfm

mod PRICED-OO-SYSTEM is
 protecting PRICED-SYSTEM .
 protecting OO-PRELUDE .

 subsort Configuration < SystemState .
endm

--- This is simply an extension of PRICED-CONF
--- to make it work within timed-oo-systems
--- the sort System is not defined in Maude
--- or Full-Maude
fmod PRICED-TIMED-SYSTEM is
 including PRICED-SYSTEM .
 including TIMED-PRELUDE .
```

```
 sort PricedTimedSystem .
 subsort ClockedSystem < PricedTimedSystem .

 --- To enable tick rules of the form
 --- {t} => {t'} in time T with cost C
 --- the following is needed
 op _with cost_ : ClockedSystem Cost
     -> PricedTimedSystem [prec 95 gather (E e)] .
endfm

fmod PRICED-MODEL-CHECKER is
  including TIMED-MODEL-CHECKER .
  including PRICED-SYSTEM .
endfm

mod PRICED-TIMED-OO-SYSTEM is
 including LTIME-INF .
 including PRICED-OO-SYSTEM .
 including TIMED-OO-PRELUDE .
 including PRICED-TIMED-SYSTEM .

 var R : Time .
 vars NeC NeC' : NEConfiguration .
 var C : Cost .

 op delta : PricedSystem Time
           -> PricedSystem [frozen (1)] .

 eq delta(NeC with cost C, R) = delta(NeC, R) with cost C .
 eq delta(NeC NeC', R) = delta(NeC, R) delta(NeC', R) .
 eq delta(none, R) = none .

 op mte : PricedSystem -> TimeInf [frozen (1)] .
 eq mte(NeC with cost C) = mte(NeC) .
 eq mte(NeC NeC') =
    minimum(mte(NeC), mte(NeC')) .
 eq mte(none) = INF .


 op rate : PricedSystem -> Cost [frozen (1)] .
 eq rate(NeC with cost C) = rate(NeC) .
 eq rate(NeC NeC') =
    rate(NeC) pluss rate(NeC') .
 eq rate(none) = free .
endm

*** This module is now included into in TIMED-MODULE-SYNTAX
fmod PRICED-MODULE-SYNTAX is
  including VIEWS .

  *** UnTimed Priced and Priced-OO modules and theories:
  op pmod_is_endpm : @Interface@ @SDeclList@ -> @Module@ .
  op pomod_is_endpom : @Interface@ @SDeclList@ -> @Module@ .

  *** Priced-Timed modules and theories:
  op ptmod_is_endptm : @Interface@ @SDeclList@ -> @Module@ .
  op ptth_is_endptth : @Interface@ @SDeclList@ -> @Module@ .

  *** Object-oriented priced-timed modules and theories:
  op ptomod_is_endptom : @Interface@ @ODeclList@ -> @Module@ .
  op ptoth_is_endptoth : @Interface@ @ODeclList@ -> @Module@ .
endfm

*** define user level PTM commands
*** imported into REAL-TIME-MAUDE-SYNTAX
fmod PTM-COMMAND-SYNTAX is
  including COMMANDS .

  *** PRICED-TIMED MAUDE commands
  *** the priced timed search command, last bubble is cost limit
  *** no limit can easily be implemented too
  op ptsearch_=>*_in time <_with cost <_. : @Bubble@ @Bubble@
                                           @Bubble@ @Bubble@
                                           -> @Command@ .
  op ptsearch_=>*_in time <_with cost <=_. : @Bubble@ @Bubble@
                                            @Bubble@ @Bubble@
                                            -> @Command@ .
  op ptsearch_=>*_in time <=_with cost <_. : @Bubble@ @Bubble@
                                            @Bubble@ @Bubble@
                                            -> @Command@ .
  op ptsearch_=>*_in time <=_with cost <=_. : @Bubble@ @Bubble@
                                             @Bubble@ @Bubble@
                                             -> @Command@ .

  op ptsearch_=>!_in time <_with cost <_. : @Bubble@ @Bubble@
                                            @Bubble@ @Bubble@
                                            -> @Command@ .
    op ptsearch_=>!_in time <_with cost <=_. : @Bubble@ @Bubble@
                                              @Bubble@ @Bubble@
                                              -> @Command@ .
    op ptsearch_=>!_in time <=_with cost <_. : @Bubble@ @Bubble@
                                              @Bubble@ @Bubble@
                                              -> @Command@ .
    op ptsearch_=>!_in time <=_with cost <=_. : @Bubble@ @Bubble@
```

```
                                          @Bubble@ @Bubble@
                                        -> @Command@ .
*** priced-timed search with just a cost limit
op ptsearch_=>*_with no time limit with cost <_. : @Bubble@
                                                   @Bubble@
                                                   @Bubble@
                                                   -> @Command@ .
op ptsearch_=>*_with no time limit with cost <=_. : @Bubble@
                                                    @Bubble@
                                                    @Bubble@
                                                    -> @Command@ .
op ptsearch_=>!_with no time limit with cost <_. : @Bubble@
                                                   @Bubble@
                                                   @Bubble@
                                                   -> @Command@ .
op ptsearch_=>!_with no time limit with cost <=_. : @Bubble@
                                                    @Bubble@
                                                    @Bubble@
                                                    -> @Command@ .
*** priced-timed search with just a time limit
*** can use parse path of find cheapest with time limit
op ptsearch_=>*_in time <_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                               -> @Command@ .
op ptsearch_=>*_in time <=_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                                -> @Command@ .

op ptsearch_=>!_in time <_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                               -> @Command@ .
op ptsearch_=>!_in time <=_with no cost limit. : @Bubble@ @Bubble@ @Bubble@
                                                -> @Command@ .

*** and finally ptsearch with no limits
op ptsearch_=>*_with no limits . : @Bubble@ @Bubble@ -> @Command@ .
op ptsearch_=>!_with no limits . : @Bubble@ @Bubble@ -> @Command@ .

*** find cheapest, finds the cheapest solution in a given time
op find cheapest_=>*_in time <_. : @Bubble@ @Bubble@ @Bubble@
                                  -> @Command@ .
op find cheapest_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@
                                   -> @Command@ .
*** find cheapest with no limits
op find cheapest_=>*_with no time limit. : @Bubble@ @Bubble@
                                          -> @Command@ .

*** find cheapest, finds the cheapest solution in a given time
op binary find cheapest_=>*_in time <_. : @Bubble@ @Bubble@ @Bubble@
                                         -> @Command@ .
op binary find cheapest_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@
                                         -> @Command@ .
*** find cheapest with no limits
op binary find cheapest_=>*_with no time limit . : @Bubble@ @Bubble@
                                                   -> @Command@ .

*** fair rew
op ptfrew_in time <=_with cost <_. : @Bubble@ @Bubble@ @Bubble@
                                     -> @Command@ .
op ptfrew_in time <=_with cost <=_. : @Bubble@ @Bubble@ @Bubble@
                                      -> @Command@ .
op ptfrew_in time <_with cost <_. : @Bubble@ @Bubble@ @Bubble@
                                    -> @Command@ .
op ptfrew_in time <_with cost <=_. : @Bubble@ @Bubble@ @Bubble@
                                     -> @Command@ .

*** without cost limit
op ptfrew_in time <=_with no cost limit. : @Bubble@ @Bubble@
                                           -> @Command@ .
op ptfrew_in time <_with no cost limit. : @Bubble@ @Bubble@
                                         -> @Command@ .

*** without time limit
op ptfrew_with no time limit with cost <_. : @Bubble@ @Bubble@
                                             -> @Command@ .
op ptfrew_with no time limit with cost <=_. : @Bubble@ @Bubble@
                                              -> @Command@ .

*** without time and cost limit
op ptfrew_with no limits. : @Bubble@ -> @Command@ .

*** unfair rew
op ptrew_in time <=_with cost <_. : @Bubble@ @Bubble@ @Bubble@
                                    -> @Command@ .
op ptrew_in time <=_with cost <=_. : @Bubble@  @Bubble@ @Bubble@
                                     -> @Command@ .
op ptrew_in time <_with cost <_. : @Bubble@ @Bubble@ @Bubble@
                                   -> @Command@ .
op ptrew_in time <_with cost <=_. : @Bubble@ @Bubble@ @Bubble@
                                    -> @Command@ .
*** without cost limit
op ptrew_in time <=_with no cost limit. : @Bubble@ @Bubble@
                                          -> @Command@ .
op ptrew_in time <_with no cost limit. : @Bubble@ @Bubble@
                                         -> @Command@ .
*** without time limit
op ptrew_with no time limit with cost <_. : @Bubble@ @Bubble@
                                            -> @Command@ .
```

```
  op ptrew_with no time limit with cost <=_. : @Bubble@ @Bubble@
                                               -> @Command@ .
  *** without time and cost limit
  op ptrew_with no limits. : @Bubble@ -> @Command@ .

  *** some simple untimed commands
  *** TPL
  op psearch_=>*_with cost <_. : @Bubble@ @Bubble@
                                 @Bubble@ -> @Command@ .

  op psearch_=>*_with cost <=_. : @Bubble@ @Bubble@
                                  @Bubble@ -> @Command@ .

  op priced find earliest_=>*_with no cost limit. : @Bubble@ @Bubble@ -> @Command@ .
  op priced find earliest_=>*_with cost <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op priced find earliest_=>*_with cost <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .

  *** TL
  op prew_with cost <_. : @Bubble@  @Bubble@ -> @Command@ .
  op prew_with cost <=_. : @Bubble@  @Bubble@ -> @Command@ .
  op pfrew_with cost <_. : @Bubble@  @Bubble@ -> @Command@ .
  op pfrew_with cost <=_. : @Bubble@  @Bubble@ -> @Command@ .

  *** TPL
  op ut find cheapest_=>*_. : @Bubble@ @Bubble@
                              -> @Command@ .

  *** The following are the Real-Time Maude model checking comamnds
  *** simply modified to accept priced modules
  *** this is done by applying the pricify transomformation
  *** to them and adding a cost to the search pattern and initial term
  op pcheck_|= <>_with no time limit . : @Bubble@ @Bubble@
                                         -> @Command@ .
  op pcheck_|= <>_in time <_. : @Bubble@ @Bubble@ @Bubble@
                                -> @Command@ .
  op pcheck_|= <>_in time <=_. : @Bubble@ @Bubble@ @Bubble@
                                 -> @Command@ .

  op pcheck_|=_until_with no time limit . : @Bubble@ @Bubble@
                                           @Bubble@ -> @Command@ .
  op pcheck_|=_until_in time <_. : @Bubble@ @Bubble@ @Bubble@
                                   @Bubble@ -> @Command@ .
  op pcheck_|=_until_in time <=_. : @Bubble@ @Bubble@ @Bubble@
                                    @Bubble@ -> @Command@ .

  op pcheck_|=_untilStable_with no time limit . : @Bubble@ @Bubble@
                                                  @Bubble@
                                                  -> @Command@ .
  op pcheck_|=_untilStable_in time <_. : @Bubble@ @Bubble@ @Bubble@
                                         @Bubble@ -> @Command@ .
  op pcheck_|=_untilStable_in time <=_. : @Bubble@ @Bubble@ @Bubble@
                                          @Bubble@ -> @Command@ .

  op pmc_|=u_. : @Bubble@ @Bubble@ -> @Command@ .
  op pmc_|=t_with no time limit . : @Bubble@ @Bubble@ -> @Command@ .
  op pmc_|=t_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op pmc_|=t_in time <_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .

endfm

*** Following module contains functions similar
*** to the ones found in GLOBALIZATION, but these
*** functions relate to cost.
fmod PRICIFY is
 protecting GLOBALIZATION .

 sort NoTerm .
 subsort Term < NoTerm .

 op NTerm : -> NoTerm [ctor] .

 vars T T' T'' LHS RHS : Term .
 var F : Qid .
 vars TL TL' : TermList .
 vars COND COND' : Condition .
 var M : Module .
 vars Q Q' Q'' Q''' : Qid .
 var SDL : SubsortDeclSet .
 var RL : Rule .
 var AS : AttrSet .
 var IL : ImportList .
 var SS : SortSet .
 var SSDS : SubsortDeclSet .
 var OPDS : OpDeclSet .
 var MAS : MembAxSet .
 var EQS : EquationSet .
 vars RLS RLS' : RuleSet .
 var RULE : Rule .
 var H : Header .
 var N : Nat .
 var OO : Bool .

 *** Add a given cost to a term
 op makePriced : Term Term -> Term .
 eq makePriced('_in`time_[T, T'], T'')
```

```
   = '_in`time_[makePriced(T, T'), T'] .
eq makePriced('`{_`}[T], T') =  ''`{_`}[makePriced(T, T')] .
eq makePriced('_with`cost_[T,T''], T')
= '_with`cost_[T, '_pluss_[T'',T']] .
eq makePriced(T, T') = '_with`cost_[T,T'] [owise] .

*** Another help function, it simply determines if there
*** is a 'with cost' in a term, much like
*** the function inTimeTerm.
*** Due to it's simplicity and elegance, the function
*** inTimeTerm has ben adapted for the purpose of finding
*** a 'with cost' term in this function
op withCostTerm : Module Term  -> Bool .
op withCostTerm : TermList  -> Bool .
eq withCostTerm(M, T) =
   withCostTerm(T) and (leastSort(M, T) == 'PricedSystem
                or leastSort(M, T) == 'PricedTimedSystem) .

*** we need an extra layer of checks to remove the in_time term
*** andd the {_} part
*** This may not be safe?
eq withCostTerm(F[TL])
   = if F == '_with`cost_ then
      true
      else
       withCostTerm(TL)
      fi .

ceq withCostTerm((T, TL)) = withCostTerm(T) or withCostTerm(TL)
if TL =/= empty .

eq withCostTerm(T) = false [owise] .

 *** Aux function that determines if a rule is a priced rule
  op pricedRule : Rule -> Bool .
   eq pricedRule(crl LHS => RHS if COND [AS] .) =
      pricedRule(rl LHS => RHS [AS] .) .

   eq pricedRule(rl LHS => RHS [AS] .)
    = withCostTerm(RHS) .

*** This aux function figures out if one sort is a subsort of another
*** This is basically equal to finding the path in a graph
*** if there's a path between Q and Q',
*** then Q is a subsort of Q', othewise it isn't
*** There may be a more elegant method of doing this,
*** or this has already been done.
*** But the operator :: fails to parse
*** if the RHS is a non-existing sort.
op subsortOf : Qid Qid Module -> Bool .
op subsortOf : Qid Qid SubsortDeclSet -> Bool .
eq subsortOf(Q, Q', M) = subsortOf(Q, Q', getSubsorts(M)) .
eq subsortOf(Q, Q', subsort Q < Q' . SDL) = true .
ceq subsortOf(Q, Q', subsort Q < Q'' . SDL)
    = subsortOf(Q'', Q', SDL) if Q' =/= Q'' .
eq subsortOf(Q, Q', none) = false .
eq subsortOf(Q, Q', subsort Q'' < Q''' . SDL) = false [owise] .

*** Extract the cost from a term
op costPart : NoTerm -> NoTerm .
op costPart : Module Term -> Term .
eq costPart(M, T) = costPart(getTerm(metaReduce(M, T))) .
op costPart : Term -> Term .
eq costPart('_in`time_[T, T']) = costPart(T) .
eq costPart('`{_`}[T]) = costPart(T) .
---- now the cost is 0 if T has leastsort =/= PricedSystem
eq costPart('_with`cost_[T,T']) = T' .
eq costPart(NTerm) = 'infcost.CostInf .



 *** Now an aux function to make sure the whole systemis contained
 *** on the left hand side (LHS) of a rule.
 *** For tick rules we just need to add cost,
 *** for instantanious priced rules  we need to add a cost and
 *** {_} to contain the whole system, if Configuration :: SystemState
 *** we also need to add a variable of sort Configuration.
 op pricifyTerm : Bool Term Module -> Term .

 *** If we do not find {_}, we add one and the rest of the system,
 *** and add a with cost term inside.
 *** if we find a {_} at the top level of the term we simply add
 *** a cost to it
 eq pricifyTerm(true, T, M)  = makePriced(T, newCostVar(T, 'OLDCOST)) .
 ceq pricifyTerm(false, T, M)
     = if OO then
        if withCostTerm(T) then
         makePriced(makePriced(''{_`}['__[Q',stripCost(T)]],
                    Q), costPart(T))
        else
         makePriced(''{_`}['__[Q',T]], Q)
        fi
       else
        makePriced(''{_`}[T], Q)
       fi
```

```
      if Q := newCostVar(T, 'OLDCOST)
      /\ OO := subsortOf('Configuration, 'SystemState, M)
      /\ Q' := if OO then
                newConfVar(T, 'C)
               else
                'none fi [owise] .

*** The first part of our transformation:
*** - All priced rules are transofrmed so that
***   they contain the whole system and the old price in the LHS
***   and transfer this to the RHS
op pricifyMod : Module -> Module .
eq pricifyMod(FM:FModule) = FM:FModule .
eq pricifyMod(mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm)
  = (mod H is IL sorts SS . SSDS OPDS MAS EQS
     pricifyRls(RLS, mod H is IL sorts SS . SSDS OPDS MAS
     EQS RLS endm) endm) .

*** We now want to flatten all rules that are priced.
*** We do this by adding a with cost term in the
*** left hand side of the rule, aswell as making ure the
*** whole system state is preserved.
*** We must now make sure the rest of the system state
*** and the total cost is transferred into the new state by the
*** rule
op pricifyRls : RuleSet Module -> RuleSet .
eq pricifyRls(RL RLS, M) =
   if pricedRule(RL) or tickRule(RL) then
    pricifyRule(RL, M)
    else
    RL
    fi
    pricifyRls(RLS, M) .

eq pricifyRls(none, M) = none .

*** it is not enought to simply add cost, we also need
*** the {_} operator and the rest of the system for instantanous rules
*** tick for tick rules the cost of the system needs to be moved inside the {_}
op pricifyRule : Rule Module -> Rule .

*** Instantaneous rules
*** rl LHS => RHS with cost C .
***    =>
*** rl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost} .
ceq pricifyRule(rl LHS => '_with`cost_[RHS,T] [AS] ., M) =
    (rl pricifyTerm(false, LHS, M)
     =>
     pricifyTerm(false, '_with`cost_[RHS,T], M) [AS] .)
if not globalSystemTerm(LHS) .

*** rl LHS => RHS with cost C if COND .
***    =>
*** rl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost} if COND .
ceq pricifyRule(crl LHS => '_with`cost_[RHS,T] if COND [AS] ., M) =
     (crl pricifyTerm(false, LHS, M)
      =>
      pricifyTerm(false, '_with`cost_[RHS,T], M) if COND [AS] .)
if not globalSystemTerm(LHS) .

*** Tick rules
*** match patterns:
*** rl {LHS} => {RHS} in time T .
***    =>
*** rl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost} in time T .
eq pricifyRule(rl ''{_'}[LHS] => '_in`time_['{_'}['R:SystemState],T] [AS] ., M) =
      (rl pricifyTerm(true, ''{_'}[LHS], M)
       =>
       pricifyTerm(true, '_in`time_['{_'}['R:SystemState],T], M) [AS] .) .

*** rl {LHS} => {RHS} in time T  .
***    =>
*** rl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost } in time T .
eq pricifyRule(crl ''{_'}[LHS] => '_in`time_['{_'}[RHS],T] if COND [AS] ., M) =
       (crl pricifyTerm(true, ''{_'}[LHS], M)
        =>
        pricifyTerm(true, '_in`time_['{_'}[RHS],T], M) if COND [AS] .) .

*** rl {LHS} => {RHS} in time T with cost C .
***    =>
*** rl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost pluss C} in time T .
eq pricifyRule(rl ''{_'}[LHS] => '_with`cost_['_in`time_['{_'}[RHS],T],T'] [AS] ., M) =
        (rl pricifyTerm(true, ''{_'}[LHS], M)
         =>
         pricifyTerm(true, '_in`time_['{_'}['_with`cost_[RHS,T']],T], M) [AS] .) .


*** crl {LHS} => {RHS} in time T if COND .
***    =>
*** crl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost} in time T
*** if COND .
eq pricifyRule(crl ''{_'}[LHS] => '_in`time_['{_'}[RHS],T] if COND [AS] ., M) =
        (crl pricifyTerm(true, ''{_'}[LHS], M)
         =>
        pricifyTerm(true, '_in`time_['{_'}[RHS],T], M) if COND [AS] .) .
```

```
*** crl {LHS} => {RHS} in time T with cost C .
***    =>
*** crl {LHS with cost OLDCOST:Cost} => {RHS  with cost OLDCOST:Cost pluss C} in time T .
eq pricifyRule(crl ''{_'}[LHS] => '_with'cost_['_in'time_['''{_'}[RHS],T],T'] if COND [AS] ., M) =
         (crl pricifyTerm(true, ''{_'}[LHS], M)
          =>
         pricifyTerm(true, '_in'time_['''{_'}['_with'cost_[RHS,T']]],T], M) if COND [AS] .) .


*** Strip the with cost operator, thus the cost from a term
op stripCost : NoTerm -> NoTerm .
op stripCost : Term -> Term .
eq stripCost('_in'time_[T, T']) = '_in'time_[stripCost(T), T'] .
eq stripCost(''{_'}[T]) =  ''{_'}[stripCost(T)] .
eq stripCost('_with'cost_[T,T'']) = T .
eq stripCost(NTerm) = NTerm .
eq stripCost(T) = T [owise] .

*** Determine if 1 cost smaller than another
op cheaper : Term Term -> Bool .
eq cheaper(T, T') = downTerm('_cheaper'than_[T,T'], true) .

*** Does exactly the same as myNewVar
*** determines the next free name of a Cost var in a rule or term
op newCostVar : Rule Qid -> Variable .
op newCostVar : Term Qid -> Variable .

op newCostVar : Rule Qid Nat -> Variable .
op newCostVar : Term Qid Nat -> Variable .

ceq newCostVar(RULE, Q)
   = if Q' in vars(RULE) then
      newCostVar(RULE, Q, 1)
     else
      'TIME_ELAPSED:Time
     fi
if Q' := conc(Q, ':Cost) .

ceq newCostVar(RULE, Q, N)
   = if Q' in vars(RULE) then
      newCostVar(RULE, Q, N + 1)
     else
      Q'
     fi
if Q' := conc(index(conc(Q, '#), N), ':Cost) .

ceq newCostVar(T, Q)
   = if Q' in vars(T) then
      newCostVar(T, Q, 1)
     else
      Q'
     fi
if Q' := conc(Q, ':Cost) .

ceq newCostVar(T, Q, N)
   = if Q' in vars(T) then
      newCostVar(T, Q, N + 1)
     else
      Q'
     fi
if Q' := conc(index(conc(Q, '#), N), ':Cost) .

*** Should make a more generic function
*** Does exactly the same as myNewVar
*** determines the next free name of a Cconfiguration var in a rule or term
op newConfVar : Rule Qid -> Variable .
op newConfVar : Term Qid -> Variable .

op newConfVar : Rule Qid Nat -> Variable .
op newConfVar : Term Qid Nat -> Variable .

ceq newConfVar(RULE, Q)
   = if Q' in vars(RULE) then
      newConfVar(RULE, Q, 1)
     else
      'TIME_ELAPSED:Time
     fi
if Q' := conc(Q, ':Configuration) .

ceq newConfVar(RULE, Q, N)
   = if Q' in vars(RULE) then
      newConfVar(RULE, Q, N + 1)
     else
      Q'
     fi
if Q' := conc(index(conc(Q, '#), N), ':Configuration) .

ceq newConfVar(T, Q)
   = if Q' in vars(T) then
      newConfVar(T, Q, 1)
     else
      Q'
     fi
if Q' := conc(Q, ':Configuration) .
```

```
  ceq newConfVar(T, Q, N)
     = if Q' in vars(T) then
        newConfVar(T, Q, N + 1)
       else
        Q'
       fi
  if Q' := conc(index(conc(Q, '#), N), ':Configuration) .

  --- The term transformation \texttt{pricifyInit} injects a `\texttt{with cost free}' into an initial term
  op pricifyInit : Term -> Term .
  eq pricifyInit(T) = makePriced(T, 'free.Cost) .

  --- The term transformation \texttt{pricifyPattern} injects a `\texttt{with cost TOTAL_COST_INCURRED:Cost}' into a search pattern
  op pricifyPattern : Term -> Term .
  eq pricifyPattern(T) = makePriced(T, newCostVar(T,'TOTAL_COST_INCURRED)) .



 --- move 'with cost' from within {}
 --- to behind the {}
 op metaMoveCost : Term -> Term .
 eq metaMoveCost('_in`time_['`{_`}['_with`cost_[T,T']],T'`]) = '_with`cost_['_in`time_['`{_`}[T],T'`],T'] .
endfm

*** Second part of the transformation:
*** - Add a given cost limit to all priced rules
fmod COST-LIMIT-TRANSFORMATION is
  pr PRICIFY .

  var B : Bool .
  vars T T' LHS RHS : Term .
  vars COND COND' : Condition .
  var M : Module .
  var SDL : SubsortDeclSet .
  var RL : Rule .
  var AS : AttrSet .
  var IL : ImportList .
  var SS : SortSet .
  var SSDS : SubsortDeclSet .
  var OPDS : OpDeclSet .
  var MAS : MembAxSet .
  var EQS : EquationSet .
  vars RLS RLS' : RuleSet .
  var RULE : Rule .
  var H : Header .

 *** Aux function that generates conditions to add to a rule
 *** The first Term is the left side of the inequality
 *** The second the right side
 ops cheaperCond cheaperEqCond : Term Term -> Condition .
 eq  cheaperCond(T,T') = 'true.Bool = '_cheaper`than_[T,T'] .
 eq  cheaperEqCond(T,T')
     = 'true.Bool = '_or_['_cheaper`than_[T,T'],'_==_[T,T']] .

  *** Parameters:
  *** - Mod
  *** - true if <, false if <=
  *** - the value of the cost limit
  op costLimitMod : Module Bool Term ->  Module .

  eq costLimitMod(FM:FModule, B, T) = FM:FModule .
  eq costLimitMod(mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm,
                  B, T)
     = (mod H is IL sorts SS . SSDS OPDS MAS EQS
        costLimitRls(RLS, B, T) endm) .

  op costLimitRls : RuleSet Bool Term -> RuleSet .
  eq costLimitRls(RL RLS, B, T)
     =
     if pricedRule(RL) then
      costLimitRule(RL, B, T)
     else
      RL
     fi
     costLimitRls(RLS, B, T) .
  eq costLimitRls(none, B, T) = none .

  *** Now the actual transformation of each rule
  op costLimitRule : Rule Bool Term -> Rule .
  ceq costLimitRule(crl LHS => RHS if COND [AS] ., B, T) =
     (crl LHS => RHS if COND /\ COND' [AS] .)
     if COND' :=
       if B then
        cheaperCond(costPart(RHS), T)
       else
        cheaperEqCond(costPart(RHS), T)
       fi .

  ceq costLimitRule(rl LHS => RHS [AS] ., B, T) =
     (crl LHS => RHS if COND [AS] .)
     if COND :=
       if B then
        cheaperCond(costPart(RHS), T)
       else
```

```
        cheaperEqCond(costPart(RHS), T)
       fi .
endfm

fmod PRICIFY-PROPS is
  protecting TIMED-MODULE-TRANSFORMATIONS .

  var IL : ImportList .
  var SS : SortSet .
  var SSDS : SubsortDeclSet .
  var OPDS : OpDeclSet .
  var MAS : MembAxSet .
  var EQS : EquationSet .
  var RLS : RuleSet .
  var H : Header .

  op pricifyProperties : Module -> Module .

  eq pricifyProperties(mod H is IL sorts SS . SSDS OPDS MAS EQS RLS endm) =
                     (mod H is IL sorts SS . SSDS OPDS MAS
                     (EQS (ceq '_|=_['`{_`}['_with`cost_['S:SystemState, 'C:Cost]], 'P:Prop] = 'true.Bool
                     if '_|=_['`{_`}['S:SystemState], 'P:Prop] = 'true.Bool [none] . ))
                     RLS endm) .

  eq pricifyProperties(fmod H is IL sorts SS . SSDS OPDS MAS EQS endfm) =
                      (fmod H is IL sorts SS . SSDS OPDS MAS
                      (EQS (ceq '_|=_['`{_`}['_with`cost_['S:SystemState, 'C:Cost]], 'P:Prop] = 'true.Bool
                       if '_|=_['`{_`}['S:SystemState], 'P:Prop] = 'true.Bool [none] . ))
                      endfm) .
endfm

fmod PRICED-MODULE-TRANSFORMATIONS is
 protecting PRICIFY .
 protecting COST-LIMIT-TRANSFORMATION .
 protecting PRICIFY-PROPS .
endfm

fmod PRICED-TIMED-SEARCH is
 pr PRICED-MODULE-TRANSFORMATIONS .
 pr TIMED-META-SEARCH .

 *** This is a simple search with a time and price bound
 *** Typical: Can some state matching a pattern and condition
 *** be found within x time and y cost.
  var M : Module .
  vars INITIALSTATE SEARCHPATTERN TIMEBOUND COSTBOUND : Term .
  var CHEAPER : Bool .
  var COND : Condition .
  var R : ResultTriple? .
  var R' : Substitution? .
  var COMP : ComparisonOp .
  var TM : TickMode .
  var Q : Qid .
  var D : Bound .
  var N : Nat .
  *** Parameters:
  *** - Module,
  *** - Term, initial state
  *** - Term, search pattern
  *** - Condition
  *** - ComparisonOp, lt, le
  *** - Term, time bound
  *** - TickMode,
  *** - Bool, true = cheaper than, false cheaper than eq
  ***   (This could possibly use comparisonop too)
  *** - Term, cost bound
  op pricedTimedSearch : Module Term Term Condition Qid Bound
                         Nat ComparisonOp Term TickMode Bool Term
                         -> ResultTriple? .

  *** 1) add with cost 0 to INITIALSTATE
  *** 2) add with cost TOTAL_COST_INCURRED:Cost to SEARCHPATTERN
  *** 3) determine whether we are lookign for cheaper than or cheaper
  ***    than eq cost bound
  *** 4) Transform M with costLimitMod
  *** 5) Send all the stuff to TIMED-META-SEARCH
  eq pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN,
                       COND, Q, D, N, COMP, TIMEBOUND, TM,CHEAPER,
                       COSTBOUND)
  = timedMetaSearch(costLimitMod(pricifyMod(M),CHEAPER,COSTBOUND),
                    pricifyInit(INITIALSTATE),
                    pricifyPattern(SEARCHPATTERN),
                    COND, Q, D, N, COMP, TIMEBOUND, TM) .


  *** this version has no cost limit
  op pricedTimedSearch : Module Term Term Condition Qid Bound
                         Nat ComparisonOp Term TickMode
                          -> ResultTriple? .


   eq pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN,
                        COND, Q, D, N, COMP, TIMEBOUND, TM)
   = timedMetaSearch(pricifyMod(M), pricifyInit(INITIALSTATE), pricifyPattern(SEARCHPATTERN),
                     COND, Q, D, N, COMP, TIMEBOUND, TM) .
```

```
endfm

fmod PRICED-UNTIMED-SEARCH is
 pr PRICED-MODULE-TRANSFORMATIONS .
 pr EXPAND-OBJECT-PATTERN .

 *** This is a simple search with a time and price bound
 *** Typical: Can some state matching a pattern and condition
 *** be found within x time and y cost.
  var M : Module .
  vars INITIALSTATE SEARCHPATTERN TIMEBOUND COSTBOUND : Term .
  var CHEAPER : Bool .
  var COND : Condition .
  var R : ResultTriple? .
  var R' : Substitution? .
  var Q : Qid .
  var D : Bound .

  *** Parameters:
  *** - Module,
  *** - Term, initial state
  *** - Term, search pattern
  *** - Condition
  *** - ComparisonOp, lt, le
  *** - Term, time bound
  *** - TickMode,
  *** - Bool, true = cheaper than, false cheaper than eq
  ***   (This could possibly use comparisonop too)
  *** - Term, cost bound
  op pricedSearch : Module Term Term Condition Qid Bound
                    Bool Term
                    -> ResultTriple? .

  *** 3) determine whether we are lookign for cheaper than or cheaper
  ***    than eq cost bound
  *** 4) Transform M with costLimitMod
  *** 5) Send all the stuff to TIMED-META-SEARCH
  eq pricedSearch(M, INITIALSTATE, SEARCHPATTERN,
                  COND, Q, D, CHEAPER,
                  COSTBOUND)
  = metaSearch(costLimitMod(pricifyMod(M),CHEAPER,COSTBOUND),
               pricifyInit(INITIALSTATE),
               expandObjectPattern(M, pricifyPattern(SEARCHPATTERN)),
               COND, Q, D, 0) .



 endfm

fmod FIND-CHEAPEST is
  pr PRICED-TIMED-SEARCH .

  var M : Module .
  vars INITIALSTATE SEARCHPATTERN TIMEBOUND COSTBOUND : Term .
  var D : Bound .
  var CHEAPER : Bool .
  var COND : Condition .
  var R : ResultTriple? .
  var R' : Substitution? .
  var COMP : ComparisonOp .
  var TM : TickMode .
  vars THIS_SEARCH PREV_SEARCH : ResultTriple? .
  var Q : Qid .

  *** findCheapest takes the following arguments:
  *** - Module
  *** - Term, the initial term
  *** - Term, the search pattern
  *** - Condition, an odinary such that condition
  *** - ComparisonOp, < or <=
  *** - Term,  timebound
  *** - TickMode, current time sampling strat
  op findCheapest : Module Term Term Condition Qid Bound
                    ComparisonOp Term TickMode -> ResultTriple? .

  op findCheapest : Module Term Term Condition Qid Bound
                    ComparisonOp Term TickMode Term ResultTriple? -> ResultTriple? .

  ceq findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                   TIMEBOUND, TM)
  = if THIS_SEARCH =/= failure then
     findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                  TIMEBOUND, TM,
                  costPart(getTerm(THIS_SEARCH)),
                  THIS_SEARCH)
    else
     failure
    fi
  if THIS_SEARCH :=
      pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN,
                        COND, Q, D,
                        0, COMP, TIMEBOUND, TM, true, 'infcost.CostInf) .

  ceq findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                   TIMEBOUND, TM, COSTBOUND, PREV_SEARCH)
```

```
      = if THIS_SEARCH =/= failure then
        findCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                     TIMEBOUND, TM,
                     costPart(getTerm(THIS_SEARCH)),
                     THIS_SEARCH)
        else
         PREV_SEARCH
        fi
  if THIS_SEARCH :=
     pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                       0, COMP, TIMEBOUND, TM, true, COSTBOUND) .

endfm

fmod UNTIMED-FIND-CHEAPEST is
  pr PRICED-UNTIMED-SEARCH .

  var M : Module .
  vars INITIALSTATE SEARCHPATTERN TIMEBOUND COSTBOUND : Term .
  var D : Bound .
  var CHEAPER : Bool .
  var COND : Condition .
  var R : ResultTriple? .
  vars THIS_SEARCH PREV_SEARCH : ResultTriple? .
  var Q : Qid .

  *** findCheapest takes the following arguments:
  *** - Module
  *** - Term, the initial term
  *** - Term, the search pattern
  *** - Condition, an odinary such that condition
  op utFindCheapest : Module Term Term Condition Qid Bound
                      -> ResultTriple? .

  op utFindCheapest : Module Term Term Condition Qid Bound
                      Term ResultTriple? -> ResultTriple? .



  ceq utFindCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D)
  = if THIS_SEARCH =/= failure then
     utFindCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                    costPart(getTerm(THIS_SEARCH)),
                    THIS_SEARCH)
    else
     failure
    fi
  if THIS_SEARCH :=
      pricedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                   true, 'infcost.CostInf) .

  ceq utFindCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                     COSTBOUND, PREV_SEARCH)
      = if THIS_SEARCH =/= failure then
        utFindCheapest(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                       costPart(getTerm(THIS_SEARCH)),
                       THIS_SEARCH)
        else
         PREV_SEARCH
        fi
  if THIS_SEARCH :=
     pricedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                  true, COSTBOUND) .

endfm

*** this module provides a function
*** that finds the cheapest by using
*** a binary search technique
*** this may not work correctly
*** with rational numbers since there
*** is an infinite number of rats between two nats
fmod FIND-CHEAPEST-BINARY is
  pr PRICED-TIMED-SEARCH .
  pr FIND-CHEAPEST .

  var M : Module .
  vars T T' BEST CURRENT : Term .
  vars INITIALSTATE SEARCHPATTERN TIMEBOUND COSTBOUND : Term .
  var D : Bound .
  var CHEAPER : Bool .
  var COND : Condition .
  var R : ResultTriple? .
  vars THIS_SEARCH PREV_SEARCH : ResultTriple? .
  var Q : Qid .
  var COMP : ComparisonOp .
  var TM : TickMode .

  op metaDiv2 : Module Term -> Term .
  eq metaDiv2(M, T) = getTerm(metaReduce(M,'div2[T])) .

  ops metaAdd metaAvg : Module Term Term -> Term .
  eq metaAdd(M, T, T') = getTerm(metaReduce(M,'_pluss_[T,T'])) .
  eq metaAvg(M, T, T') = metaDiv2(M, metaAdd(M, T, T')) .
```

```
  op findCheapestBin : Module Term Term Condition Qid Bound
                       ComparisonOp Term TickMode -> ResultTriple? .

  ***op findCheapestBin : Module Term Term Condition Qid Bound
  ***                     ComparisonOp Term TickMode Term Term ResultTriple? -> ResultTriple? .

  op findCheapestBin : Module Term Term Condition Qid Bound
                       ComparisonOp Term TickMode Term ResultTriple? -> ResultTriple? .


  ceq findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                      TIMEBOUND, TM)
     = if THIS_SEARCH =/= failure then
        findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                        TIMEBOUND, TM,
                        metaAvg(M, costPart(getTerm(THIS_SEARCH)), 'free.Cost),
                        THIS_SEARCH)
       else
        failure
       fi
     if THIS_SEARCH :=
         pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                           0, COMP, TIMEBOUND, TM, true, 'infcost.CostInf) .

ceq findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                    TIMEBOUND, TM, COSTBOUND, PREV_SEARCH)
      = if THIS_SEARCH =/= failure then
         *** we search between the value of this search and 0
         findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                         TIMEBOUND, TM,
                         metaAvg(M, costPart(getTerm(THIS_SEARCH)), 'free.Cost),
                         THIS_SEARCH)
        else
         *** we search between the previous value and best known
         *** if this ends up being best known we are done
         if CURRENT == COSTBOUND then
          PREV_SEARCH
         else
          findCheapestBin(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D, COMP,
                          TIMEBOUND, TM,
                          CURRENT,
                          PREV_SEARCH)
         fi
        fi
  if CURRENT := metaAvg(M,  costPart(getTerm(PREV_SEARCH)), COSTBOUND)
     /\
     THIS_SEARCH :=
     pricedTimedSearch(M, INITIALSTATE, SEARCHPATTERN, COND, Q, D,
                       0, COMP, TIMEBOUND, TM, true, COSTBOUND) .

endfm

--- A priced rewrite may also be useful
fmod PRICED-TIMED-REWRITE is
 pr PRICED-MODULE-TRANSFORMATIONS .
 pr TIMED-META-REWRITE .

 *** This is a simple search with a time and price bound
 *** Typical: Can some state matching a pattern and condition
 *** be found within x time and y cost.
  var M : Module .
  vars INITIALSTATE SEARCHPATTERN TIMEBOUND COSTBOUND : Term .
  var BOUND : Bound .
  var CHEAPER : Bool .
  var COND : Condition .
  var R : ResultTriple? .
  var R' : Substitution? .
  var COMP : ComparisonOp .
  var TM : TickMode .
  var N : Nat .

  --- ptrew with cost limit
  op pricedTimedMetaRewrite : Module Term Bound ComparisonOp Term
                              TickMode Bool Term -> ResultPair .


  eq pricedTimedMetaRewrite(M, INITIALSTATE, BOUND, COMP,
                            TIMEBOUND,
                            TM, CHEAPER, COSTBOUND) =
     timedMetaRewrite(costLimitMod(pricifyMod(M),CHEAPER,COSTBOUND),
                      pricifyInit(INITIALSTATE),
                      BOUND,COMP, TIMEBOUND, TM) .
 --- ptfrew with cost limit
 op pricedTimedMetaFRewrite : Module Term Bound Nat
                              ComparisonOp Term TickMode
                              Bool Term -> ResultPair .

  eq pricedTimedMetaFRewrite(M, INITIALSTATE, BOUND, N, COMP,
                             TIMEBOUND,
                             TM, CHEAPER, COSTBOUND) =
     timedMetaFRewrite(costLimitMod(pricifyMod(M),CHEAPER,COSTBOUND),
                       pricifyInit(INITIALSTATE),
                       BOUND, N, COMP, TIMEBOUND, TM) .

  --- ptrew without cost limit
```

```
  op pricedTimedMetaRewrite : Module Term Bound ComparisonOp Term
                              TickMode -> ResultPair .


  eq pricedTimedMetaRewrite(M, INITIALSTATE, BOUND, COMP,
                            TIMEBOUND, TM) =
     timedMetaRewrite(pricifyMod(M),
                      pricifyInit(INITIALSTATE),
                      BOUND,COMP, TIMEBOUND, TM) .

 --- ptfrew without cost limit
 op pricedTimedMetaFRewrite : Module Term Bound Nat
                              ComparisonOp Term TickMode
                              -> ResultPair .

 eq pricedTimedMetaFRewrite(M, INITIALSTATE, BOUND, N, COMP,
                            TIMEBOUND, TM) =
     timedMetaFRewrite(pricifyMod(M),
                       pricifyInit(INITIALSTATE),
                       BOUND, N, COMP, TIMEBOUND, TM) .

endfm


fmod PRICED-UNTIMED-REWRITE is
 pr PRICED-MODULE-TRANSFORMATIONS .


  op pricedMetaRewrite : Module Term Bound Bool Term -> ResultPair .
  op pricedMetaFRewrite : Module Term Bound Nat
                          Bool Term -> ResultPair .
  *** unfair and fair rewrite!


  var M : Module .
  vars T T' : Term .
  var B : Bound .
  var CHEAPER : Bool .
  var N : Nat .


  eq pricedMetaRewrite(M, T, B, CHEAPER, T') =
     metaRewrite(
        costLimitMod(pricifyMod(M),CHEAPER,T'),
        pricifyInit(T),
        B) .

  eq pricedMetaFRewrite(M, T, B, N, CHEAPER, T') =
     metaFrewrite(
        costLimitMod(pricifyMod(M),CHEAPER,T'),
        pricifyInit(T),
        B, N) .
endfm

*** Hack!
fmod PRICED-UNIT-PROCESSING is
  --- for now this is a somewhat hacky way of
  --- adding priced modules
  pr TIMED-UNIT-PROCESSING .

  vars T T' T'' T''' : Term .
  vars F F' : Qid .
  var TL : TermList .
  var TD : TimedData .

  *** PRICED modules and theories
  eq timedPreModuleToPreModule('ptomod_is_endptom[T, T']) =
  'omod_is_endom[T, '__['including_.['token[''PRICED-TIMED-OO-SYSTEM.Qid]],
                            T']] .

  eq timedPreModuleToPreModule('ptmod_is_endptm[T, T']) =
  'mod_is_endm[T, '__['including_.['token[''PRICED-TIMED-SYSTEM.Qid]],
                            T']] .

  eq timedPreModuleToPreModule('pomod_is_endpom[T, T']) =
  'omod_is_endom[T, '__['including_.['token[''PRICED-OO-SYSTEM.Qid]],
                            T']] .

  eq timedPreModuleToPreModule('pmod_is_endpm[T, T']) =
  'mod_is_endm[T, '__['including_.['token[''PRICED-SYSTEM.Qid]],
                            T']] .

  --- PRICED mods
  ceq processTimedMetaLevel(F[T, T'], TD) =
        F[T, processTimedMetaLevel(T', TD)]
         if (F == 'ptomod_is_endptom)
         or (F == 'ptmod_is_endptm)
         or (F == 'pomod_is_endpom)
         or (F == 'pmod_is_endpm) .
endfm

*** This module simply separates out the priced commands for easier
*** overview
fmod PTM-COMMAND-PROCESSING is
  inc TIMED-COMMAND-PROCESSING .
```

```
*** PTM commands
pr FIND-CHEAPEST .
pr FIND-CHEAPEST-BINARY .
pr PRICED-TIMED-REWRITE .

*** untimed stuff do they go here?
pr PRICED-UNTIMED-REWRITE .
pr UNTIMED-FIND-CHEAPEST .

vars T T' T'' T''' T1 T2 T3 T4 T5 T6 PROPTERM TIMEBOUND
     LIMIT LIMIT'  TERM COSTLIMIT : Term .
var Q : Qid .
var ME : ModuleExpression .
var DB : Database .
vars B CHEAPER FAIR : Bool .
var ODS : OpDeclSet .
var TMB : [Tuple<Term|Module|OpDeclSet|Bound>] .
var TM : [Tuple<Term|Module|OpDeclSet>] .
var RP : [ResultPair] .
var RT : [ResultTriple] .
var TS : Termset .
var TL : TermList .
vars COMP COMP' : ComparisonOp .
vars M M' MOD : Module .
var QIL : QidList .
var D BOUND : Bound .
var VDS : OpDeclSet .
vars TiM SOLVEDTICKMODE : TickMode .
vars I J N : Nat .
vars COND COND' : Condition .
vars SEARCHPATTERN SEARCHPATTERN' : TermCondition .
var B? : [Bool] .
var R3? : ResultTriple? .

*** Parsing the input from pricedtimed execution

*** this function will only be used on qids containing
*** a cost limit
op ctComp : Qid -> Bool .
eq ctComp(Q)
   = find(string(Q), "with`cost`<=", 0) == notFound .

*** For now this is basically the same as procTimedCommand,
*** but keeping timed and priced-timed stuff apart seems cleaner
op procPriceTimedCommand : Term ModuleExpression Database TickMode
                           -> QidList .

*** find cheapest init, pattern, timelimit
ceq procPriceTimedCommand(Q[T, T', T''], ME, DB, TiM) =
   (if compiledModule(ME, DB) then
     preprocessTimedCommandTPL(ME, getFlatModule(ME, DB), unbounded, getVars(ME, DB), DB, Q, T, T', T'',TiM)
    else
     preprocessTimedCommandTPL(
      modExp(evalModExp(ME, DB)),
      getFlatModule(modExp(evalModExp(ME,
                                      DB)),
                    database(evalModExp(ME, DB))),
      unbounded,
      getVars(modExp(evalModExp(ME, DB)),database(evalModExp(ME,
                                                             DB))),
     database(evalModExp(ME, DB)),
     Q, T, T', T'', TiM)
    fi)
if (find(string(Q), "in`time`<", 0) =/= notFound and
    (find(string(Q), "find`cheapest", 0) =/= notFound)) or
    --- added to catch ptsearch with not cost limit
    ((find(string(Q), "ptsearch", 0) =/= notFound) and
    (find(string(Q), "with`no`cost`limit", 0) =/= notFound)) .

*** untimed priced search with cost limit
ceq procPriceTimedCommand(Q[T, T', T''], ME, DB, TiM) =
   (if compiledModule(ME, DB) then
     preprocessPricedTPB(ME, getFlatModule(ME, DB),
     unbounded, getVars(ME, DB), DB, Q, T, T', T'')
    else
     preprocessPricedTPB(modExp(evalModExp(ME, DB)),
      getFlatModule(modExp(evalModExp(ME,
                                      DB)),
                    database(evalModExp(ME, DB))),
      unbounded,
      getVars(modExp(evalModExp(ME, DB)),database(evalModExp(ME,
                                                             DB))),
     database(evalModExp(ME, DB)),
     Q, T, T', T'')
    fi)
if  (find(string(Q), "with`cost", 0) =/= notFound and
    (find(string(Q), "psearch", 0) =/= notFound)) .

*** priced timed search with time and cost limits
ceq procPriceTimedCommand(Q[T, T', T'', T'''], ME, DB, TiM) =
     (if compiledModule(ME, DB) then
       preprocessPTimedTPLB(ME, getFlatModule(ME, DB),
                                unbounded, getVars(ME, DB), DB,
                                Q, T, T', T'', T''', TiM)
      else
```

```
      preprocessPTimedTPLB(modExp(evalModExp(ME, DB)),
                                getFlatModule(modExp(
                                evalModExp(ME, DB)),
                                database(evalModExp(ME, DB))),
                                unbounded,
                                getVars(modExp(
                                          evalModExp(ME,DB)),
                                database(evalModExp(ME,DB))),
                                database(evalModExp(ME, DB)),
                                Q, T, T', T'', T''', TiM)
     fi)
 if (find(string(Q), "in`time", 0) =/= notFound) and
    (find(string(Q), "ptsearch", 0) =/= notFound) and
    --- added to not catch with no cost limit
    (find(string(Q), "with`cost", 0) =/= notFound) .

*** priced timed search with no time limit, but cost limit
ceq procPriceTimedCommand(Q[T, T', T''], ME, DB, TiM) =
    (if compiledModule(ME, DB) then
      preprocessPTimedTPB(ME, getFlatModule(ME, DB),
                                unbounded, getVars(ME, DB), DB,
                                Q, T, T', T'', TiM)
     else
      preprocessPTimedTPB(modExp(evalModExp(ME, DB)),
                                getFlatModule(modExp(
                                evalModExp(ME, DB)),
                                database(evalModExp(ME, DB))),
                                unbounded,
                                getVars(modExp(
                                          evalModExp(ME,DB)),
                                database(evalModExp(ME,DB))),
                                database(evalModExp(ME, DB)),
                                Q, T, T', T'', TiM)
     fi)
 if ((find(string(Q), "with`no`time", 0) =/= notFound) and
    (find(string(Q), "ptsearch", 0) =/= notFound)) or
    ((find(string(Q), "find`earliest", 0) =/= notFound) and
    (find(string(Q), "with`cost`<", 0) =/= notFound)) .

 *** ptfrew with time and cost limits
 ceq procPriceTimedCommand(Q[T, T', T''], ME, DB, TiM) =
      (if compiledModule(ME, DB) then
       preprocessPTimedTLB(ME, getFlatModule(ME, DB),
                                  unbounded, getVars(ME, DB), DB,
                                  Q, T, T', T'', TiM)
       else
        preprocessPTimedTLB(modExp(evalModExp(ME, DB)),
                                  getFlatModule(modExp(
                                  evalModExp(ME, DB)),
                                  database(evalModExp(ME, DB))),
                                  unbounded,
                                  getVars(modExp(
                                            evalModExp(ME,DB)),
                                  database(evalModExp(ME,DB))),
                                  database(evalModExp(ME, DB)),
                                  Q, T, T', T'', TiM)
       fi)
if ((find(string(Q), "ptfrew_in`time", 0) =/= notFound) or
   (find(string(Q), "ptrew_in`time", 0) =/= notFound)) and
   (find(string(Q), "with`cost", 0) =/= notFound) .

 *** ptfrew with no timelimit, but costlimit
 ceq procPriceTimedCommand(Q[T, T'], ME, DB, TiM) =
      (if compiledModule(ME, DB) then
       preprocessPTimedTB(ME, getFlatModule(ME, DB),
                                  unbounded, getVars(ME, DB), DB,
                                  Q, T, T', TiM)
       else
        preprocessPTimedTB(modExp(evalModExp(ME, DB)),
                                  getFlatModule(modExp(
                                  evalModExp(ME, DB)),
                                  database(evalModExp(ME, DB))),
                                  unbounded,
                                  getVars(modExp(
                                            evalModExp(ME,DB)),
                                  database(evalModExp(ME,DB))),
                                  database(evalModExp(ME, DB)),
                                  Q, T, T', TiM)
       fi)
if ((find(string(Q), "ptfrew_with`no`time", 0) =/= notFound) or
   (find(string(Q), "ptrew_with`no`time", 0) =/= notFound) or
   (find(string(Q), "pfrew", 0) =/= notFound) or
   (find(string(Q), "prew", 0) =/= notFound)) and
   (find(string(Q), "with`cost", 0) =/= notFound) .

*** ptfrew with time limit and no cost limit
*** preprocessPTimedTL
ceq procPriceTimedCommand(Q[T, T'], ME, DB, TiM) =
      (if compiledModule(ME, DB) then
       preprocessPTimedTL(ME, getFlatModule(ME, DB),
                                  unbounded, getVars(ME, DB), DB,
                                  Q, T, T', TiM)
       else
        preprocessPTimedTL(modExp(evalModExp(ME, DB)),
                                  getFlatModule(modExp(
```

```
                                    evalModExp(ME, DB)),
                                    database(evalModExp(ME, DB))),
                                    unbounded,
                                    getVars(modExp(
                                             evalModExp(ME,DB)),
                                    database(evalModExp(ME,DB))),
                                    database(evalModExp(ME, DB)),
                                    Q, T, T', TiM)
        fi)
 if ((find(string(Q), "ptfrew_in`time", 0) =/= notFound) or
    (find(string(Q), "ptrew_in`time", 0) =/= notFound)) and
    (find(string(Q), "with`no`cost", 0) =/= notFound) .

 *** ptfrew with no time limit and no cost limit
 *** preprocessPTimedT
 ceq procPriceTimedCommand(Q[T], ME, DB, TiM) =
       (if compiledModule(ME, DB) then
         preprocessPTimedT(ME, getFlatModule(ME, DB),
                           unbounded, getVars(ME, DB), DB,
                           Q, T, TiM)
        else
         preprocessPTimedT(modExp(evalModExp(ME, DB)),
                                    getFlatModule(modExp(
                                    evalModExp(ME, DB)),
                                    database(evalModExp(ME, DB))),
                                    unbounded,
                                    getVars(modExp(
                                             evalModExp(ME,DB)),
                                    database(evalModExp(ME,DB))),
                                    database(evalModExp(ME, DB)),
                                    Q, T, TiM)
        fi)
 if (find(string(Q), "ptfrew_with`no`limits", 0) =/= notFound) or
    (find(string(Q), "ptrew_with`no`limits", 0) =/= notFound) .

*** find cheapest and ptsearch with no limits
ceq procPriceTimedCommand(Q[T, T'], ME, DB, TiM) =
    (if compiledModule(ME, DB) then
      preprocessPTimedTP(ME, getFlatModule(ME, DB),
                         unbounded, getVars(ME, DB), DB,
                         Q, T, T', TiM)
     else
      preprocessPTimedTP(modExp(evalModExp(ME, DB)),
                                getFlatModule(modExp(
                                evalModExp(ME, DB)),
                                database(evalModExp(ME, DB))),
                                unbounded,
                                getVars(modExp(evalModExp(ME,DB)),
                                database(evalModExp(ME,DB))),
                                database(evalModExp(ME, DB)),
                                Q, T, T', TiM)
     fi)
 if ((find(string(Q), "with`no`time", 0) =/= notFound) and
    (find(string(Q), "find`cheapest", 0) =/= notFound)) or
    ((find(string(Q), "with`no`limits", 0) =/= notFound) and
    (find(string(Q), "ptsearch", 0) =/= notFound)) or
    (find(string(Q), "find`earliest_=>*_with`no`cost`limit", 0) =/= notFound) or
    *** untimed stuff
    (find(string(Q), "ut`find`cheapest", 0) =/= notFound) .

*** PREPROCESSING

*** this is essentially a slightly modified of
*** preprocessTimedCommandTPCLCL
op preprocessPTimedTP : ModuleExpression Module Bound
                        OpDeclSet Database Qid Term Term TickMode
                        -> QidList .

*** bad init term
ceq preprocessPTimedTP(ME, M, D, VDS, DB, Q, T1, T2, TiM) =
        ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
         'does 'not 'parse. '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                      Tuple<Term|Module|OpDeclSet|Bound> .


 *** bad search pattern
 ceq preprocessPTimedTP(ME, M, D, VDS, DB, Q, T1, T2, TiM) =
        ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Search 'pattern
         'does 'not 'parse 'in 'module
         '\o eMetaPrettyPrint(getName(MOD)) '\s '\c  '. '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                      VDS, DB)
        /\  not searchPattern(TERM, T2, MOD, B, getVars(getName(MOD),
                                                        DB),
                              DB) :: TermCondition .



 *** bad tick mode
 ceq preprocessPTimedTP(ME, M, D, VDS, DB, Q,T1, T2, TiM) =
```

```
         ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
          printTickAmount(TiM)
          'does 'not 'parse 'in 'module
          '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
       if B := included('META-MODULE,
                        getImports(getTopModule(ME, DB)), DB)
          /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                         D, VDS, DB)
          /\  searchPattern(TERM, T2, MOD, B,
                            getVars(getName(MOD), DB),
                            DB) :: TermCondition
          /\  not solveTickMode(TiM, MOD, B,
                                getVars(getName(MOD), DB),
                                DB) :: TickMode .

 eq preprocessPTimedTP(ME, unitError(QIL), D, VDS, DB, Q,
                                T1, T2, TiM) =
      qidError(QIL) .


 *** now for one that actually works!
 ceq preprocessPTimedTP(ME, M, D, VDS, DB, Q, T1, T2, TiM) =
     procParsedPTimedCommandTP(Q, MOD, TERM, BOUND,
                               getTerm(metaReduce(MOD,
                               termPart(SEARCHPATTERN))),
                               condPart(SEARCHPATTERN),
                               SOLVEDTICKMODE)
     if B := included('META-MODULE, getImports(getTopModule(ME, DB)),
                      DB)
        /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
        /\  SEARCHPATTERN := searchPattern(TERM, T2, MOD, B,
                                           getVars(getName(MOD), DB), DB)
        /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                            getVars(getName(MOD), DB), DB) .
*** for priced untimed search
*** preprocessTimedCommandTPCLCL
op preprocessPricedTPB : ModuleExpression Module Bound
                              OpDeclSet Database Qid Term Term Term -> QidList .

*** bad init term
ceq preprocessPricedTPB(ME, M, D, VDS, DB, Q, T1, T2,
                                 T6) =
        ('\n '\r 'Error '\c 'in 'priced 'command:  'Command/module/initterm
         'does 'not 'parse. '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                      Tuple<Term|Module|OpDeclSet|Bound> .


 *** bad search pattern
 ceq preprocessPricedTPB(ME, M, D, VDS, DB, Q, T1, T2,
                                  T6) =
        ('\n '\r 'Error '\c 'in 'priced 'command:  'Search 'pattern
         'does 'not 'parse 'in 'module
         '\o eMetaPrettyPrint(getName(MOD)) '\s '\c  '. '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
        /\  not searchPattern(TERM, T2, MOD, B, getVars(getName(MOD),
                                                        DB),
                              DB) :: TermCondition .


 *** bad cost limit
 ceq preprocessPricedTPB(ME, M, D, VDS, DB, Q, T1, T2, T6) =
        ('\n '\r 'Error '\c 'in 'priced 'command:  'Cost 'limit 'term
         'does 'not 'parse 'in 'module
         '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                      VDS, DB)
        /\ searchPattern(TERM, T2, MOD, B, getVars(getName(MOD), DB),
                         DB) :: TermCondition
        /\ not
           (solveBubbles(T6, MOD, B,
                         getVars(getName(MOD),DB), DB) :: Term) .


 eq preprocessPricedTPB(ME, unitError(QIL), D, VDS, DB, Q,
                                T1, T2, T6) =
      qidError(QIL) .


 *** now for one that actually works!
 ceq preprocessPricedTPB(ME, M, D, VDS, DB, Q, T1, T2, T6) =
        procParsedPricedTPB(Q, MOD, TERM, BOUND,
                            getTerm(metaReduce(MOD,
                            termPart(SEARCHPATTERN))),
                             condPart(SEARCHPATTERN),
                             CHEAPER, COSTLIMIT)
     if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
```

```
       /\ CHEAPER := ctComp(Q)

       /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                      VDS, DB)
       /\  SEARCHPATTERN := searchPattern(TERM, T2, MOD, B,
                                          getVars(getName(MOD), DB), DB)
       /\  COSTLIMIT := solveBubbles(T6, MOD, B,
                                     getVars(getName(MOD),DB), DB) .

*** this is essentially a slightly modified of
*** preprocessTimedCommandTPCLCL
op preprocessPTimedTPLB : ModuleExpression Module Bound
                          OpDeclSet Database Qid Term Term Term Term TickMode
                          -> QidList .

*** bad init term
ceq preprocessPTimedTPLB(ME, M, D, VDS, DB, Q, T1, T2, T4,
                         T6, TiM) =
       ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
        'does 'not 'parse. '\o '\n)
    if B := included('META-MODULE,
                     getImports(getTopModule(ME, DB)), DB)
       /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                     Tuple<Term|Module|OpDeclSet|Bound> .


*** bad search pattern
ceq preprocessPTimedTPLB(ME, M, D, VDS, DB, Q, T1, T2, T4,
                         T6, TiM) =
       ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Search 'pattern
        'does 'not 'parse 'in 'module
        '\o eMetaPrettyPrint(getName(MOD)) '\s '\c  '. '\o '\n)
    if B := included('META-MODULE,
                     getImports(getTopModule(ME, DB)), DB)
       /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                     VDS, DB)
       /\  not searchPattern(TERM, T2, MOD, B, getVars(getName(MOD),
                                                       DB),
                             DB) :: TermCondition .

*** bad time limit
ceq preprocessPTimedTPLB(ME, M, D, VDS, DB, Q, T1, T2,
                         T4, T6, TiM) =
       ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Time 'limit 'term
        'does 'not 'parse 'in 'module
        '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
    if B := included('META-MODULE,
                     getImports(getTopModule(ME, DB)), DB)
       /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                     VDS, DB)
       /\ searchPattern(TERM, T2, MOD, B, getVars(getName(MOD), DB),
                        DB) :: TermCondition
       /\ not
          (solveBubbles(T4, MOD, B, getVars(getName(MOD), DB), DB) :: Term) .

*** bad cost limit
ceq preprocessPTimedTPLB(ME, M, D, VDS, DB, Q, T1, T2,
                           T4, T6, TiM) =
       ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Cost 'limit 'term
        'does 'not 'parse 'in 'module
        '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
    if B := included('META-MODULE,
                     getImports(getTopModule(ME, DB)), DB)
       /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                     VDS, DB)
       /\ searchPattern(TERM, T2, MOD, B, getVars(getName(MOD), DB),
                        DB) :: TermCondition
       /\ not
          (solveBubbles(T6, MOD, B,
                        getVars(getName(MOD),DB), DB) :: Term) .

*** bad tick mode
ceq preprocessPTimedTPLB(ME, M, D, VDS, DB, Q,T1, T2, T4, T6,
                         TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
          printTickAmount(TiM)
          'does 'not 'parse 'in 'module
          '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                        D, VDS, DB)
         /\  searchPattern(TERM, T2, MOD, B,
                           getVars(getName(MOD), DB),
                           DB) :: TermCondition
         /\  solveBubbles(T4, MOD, B,
                          getVars(getName(MOD), DB), DB) :: Term
         /\  solveBubbles(T6, MOD, B,
                          getVars(getName(MOD), DB), DB) :: Term
         /\  not solveTickMode(TiM, MOD, B,
                               getVars(getName(MOD), DB),
                               DB) :: TickMode .

eq preprocessPTimedTPLB(ME, unitError(QIL), D, VDS, DB, Q,
                                T1, T2, T4, T6, TiM) =
```

```
     qidError(QIL) .


 *** now for one that actually works!
 ceq preprocessPTimedTPLB(ME, M, D, VDS, DB, Q, T1, T2, T4,
                                T6, TiM) =
       procParsedPTimedCommandTPLB(Q, MOD, TERM, BOUND,
                                   getTerm(metaReduce(MOD,
                                    termPart(SEARCHPATTERN))),
                                   condPart(SEARCHPATTERN),
                                   COMP, LIMIT, CHEAPER, COSTLIMIT,
                                   SOLVEDTICKMODE)
     if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
        /\ CHEAPER := ctComp(Q)
        /\ COMP :=  commandToComp(Q)
        /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
        /\  SEARCHPATTERN := searchPattern(TERM, T2, MOD, B,
                                           getVars(getName(MOD), DB), DB)

        /\  LIMIT := solveBubbles(T4, MOD, B,
                                  getVars(getName(MOD), DB), DB)

        /\  COSTLIMIT := solveBubbles(T6, MOD, B,
                                      getVars(getName(MOD),DB), DB)
        /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                            getVars(getName(MOD), DB), DB) .

 *** preprocessPTimedTPB
 *** used for: ptsearch with no time limit cheaper than (or eq)
 *** preprocessTimedCommandTPCLCL
 op preprocessPTimedTPB : ModuleExpression Module Bound
                          OpDeclSet Database Qid Term Term Term TickMode -> QidList .

 *** bad init term
 ceq preprocessPTimedTPB(ME, M, D, VDS, DB, Q, T1, T2, T6, TiM) =
        ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
         'does 'not 'parse. '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                      Tuple<Term|Module|OpDeclSet|Bound> .


 *** bad search pattern
 ceq preprocessPTimedTPB(ME, M, D, VDS, DB, Q, T1, T2, T6, TiM) =
        ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Search 'pattern
         'does 'not 'parse 'in 'module
         '\o eMetaPrettyPrint(getName(MOD)) '\s '\c  '. '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
        /\  not searchPattern(TERM, T2, MOD, B, getVars(getName(MOD),
                                                        DB),
                              DB) :: TermCondition .


 *** bad cost limit
 ceq preprocessPTimedTPB(ME, M, D, VDS, DB, Q, T1, T2, T6, TiM) =
        ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Cost 'limit 'term
         'does 'not 'parse 'in 'module
         '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
     if B := included('META-MODULE,
                      getImports(getTopModule(ME, DB)), DB)
        /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                      VDS, DB)
        /\ searchPattern(TERM, T2, MOD, B, getVars(getName(MOD)), DB),
                         DB) :: TermCondition
        /\ not
           (solveBubbles(T6, MOD, B,
                         getVars(getName(MOD),DB), DB) :: Term) .

 *** bad tick mode
 ceq preprocessPTimedTPB(ME, M, D, VDS, DB, Q,T1, T2, T6, TiM) =
          ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
           printTickAmount(TiM)
           'does 'not 'parse 'in 'module
           '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
       if B := included('META-MODULE,
                        getImports(getTopModule(ME, DB)), DB)
          /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                         D, VDS, DB)
          /\  searchPattern(TERM, T2, MOD, B,
                            getVars(getName(MOD), DB),
                            DB) :: TermCondition
          /\  solveBubbles(T6, MOD, B,
                           getVars(getName(MOD), DB), DB) :: Term
          /\  not solveTickMode(TiM, MOD, B,
                                getVars(getName(MOD), DB),
                                DB) :: TickMode .

 eq preprocessPTimedTPB(ME, unitError(QIL), D, VDS, DB, Q,
                                T1, T2, T6, TiM) =
      qidError(QIL) .
```

```
*** now for one that actually works!
ceq preprocessPTimedTPB(ME, M, D, VDS, DB, Q, T1, T2, T6, TiM) =
    procParsedPTimedCommandTPB(Q, MOD, TERM, BOUND,
                               getTerm(metaReduce(MOD,
                                termPart(SEARCHPATTERN))),
                               condPart(SEARCHPATTERN),
                               CHEAPER, COSTLIMIT,
                               SOLVEDTICKMODE)
    if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
       /\ CHEAPER := ctComp(Q)
       /\  {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                      VDS, DB)
       /\  SEARCHPATTERN := searchPattern(TERM, T2, MOD, B,
                                          getVars(getName(MOD), DB), DB)
       /\  COSTLIMIT := solveBubbles(T6, MOD, B,
                                     getVars(getName(MOD),DB), DB)
       /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                           getVars(getName(MOD), DB), DB) .


 *** price-timed rewrites with term, timelimit, costlimit
 *** this is essentially a slightly modified of
 *** preprocessTimedCommandTPCLCL
 op preprocessPTimedTLB : ModuleExpression Module Bound
                          OpDeclSet Database Qid Term Term Term TickMode -> QidList .

 *** bad init term
 ceq preprocessPTimedTLB(ME, M, D, VDS, DB, Q, T1, T4,
                                    T6, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
          'does 'not 'parse. '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                       Tuple<Term|Module|OpDeclSet|Bound> .



  *** bad time limit
  ceq preprocessPTimedTLB(ME, M, D, VDS, DB, Q, T1,
                                  T4, T6, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Time 'limit 'term
          'does 'not 'parse 'in 'module
          '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
         /\ not
            (solveBubbles(T4, MOD, B, getVars(getName(MOD), DB), DB) :: Term) .

  *** bad cost limit
  ceq preprocessPTimedTLB(ME, M, D, VDS, DB, Q, T1,
                          T4, T6, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Cost 'limit 'term
          'does 'not 'parse 'in 'module
          '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
         /\ not
            (solveBubbles(T6, MOD, B,
                          getVars(getName(MOD),DB), DB) :: Term) .

*** bad tick mode
ceq preprocessPTimedTLB(ME, M, D, VDS, DB, Q,T1, T4, T6,
                        TiM) =
    ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
    printTickAmount(TiM)
           'does 'not 'parse 'in 'module
           '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
       if B := included('META-MODULE,
                        getImports(getTopModule(ME, DB)), DB)
          /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                         D, VDS, DB)
          /\  solveBubbles(T4, MOD, B,
                           getVars(getName(MOD), DB), DB) :: Term
          /\  solveBubbles(T6, MOD, B,
                           getVars(getName(MOD), DB), DB) :: Term
          /\  not solveTickMode(TiM, MOD, B,
                                getVars(getName(MOD), DB),
                                DB) :: TickMode .

eq preprocessPTimedTLB(ME, unitError(QIL), D, VDS, DB, Q,
                       T1, T4, T6, TiM) =
   qidError(QIL) .


*** now for one that actually works!
ceq preprocessPTimedTLB(ME, M, D, VDS, DB, Q, T1, T4,  T6, TiM) =
    procParsedPTimedCommandTLB(Q, MOD, TERM, BOUND, COMP, LIMIT,
                               CHEAPER, COSTLIMIT, SOLVEDTICKMODE)
```

```
if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
   /\ CHEAPER := ctComp(Q)
   /\ COMP :=  commandToComp(Q)
   /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D, VDS,
                                                 DB)
   /\  LIMIT := solveBubbles(T4, MOD, B, getVars(getName(MOD), DB),
                             DB)
   /\  COSTLIMIT := solveBubbles(T6, MOD, B,
                                 getVars(getName(MOD),DB), DB)
   /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                       getVars(getName(MOD), DB), DB) .

 *** price-timed rewrites with term, no timelimit, but a costlimit
 *** this is essentially a slightly modified of
 *** preprocessTimedCommandTPCLCL
 op preprocessPTimedTB : ModuleExpression Module Bound
                         OpDeclSet Database Qid Term Term TickMode -> QidList .

 *** bad init term
 ceq preprocessPTimedTB(ME, M, D, VDS, DB, Q, T1, T6, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
          'does 'not 'parse. '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                       Tuple<Term|Module|OpDeclSet|Bound> .

  *** bad cost limit
  ceq preprocessPTimedTB(ME, M, D, VDS, DB, Q, T1, T6, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Cost 'limit 'term
          'does 'not 'parse 'in 'module
          '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
         /\ not
            (solveBubbles(T6, MOD, B,
                          getVars(getName(MOD),DB), DB) :: Term) .

*** bad tick mode
ceq preprocessPTimedTB(ME, M, D, VDS, DB, Q,T1, T6, TiM) =
    ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
    printTickAmount(TiM)
           'does 'not 'parse 'in 'module
           '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
       if B := included('META-MODULE,
                        getImports(getTopModule(ME, DB)), DB)
          /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                         D, VDS, DB)
          /\  solveBubbles(T6, MOD, B,
                           getVars(getName(MOD), DB), DB) :: Term
          /\  not solveTickMode(TiM, MOD, B,
                                getVars(getName(MOD), DB),
                                DB) :: TickMode .

eq preprocessPTimedTB(ME, unitError(QIL), D, VDS, DB, Q,
                      T1, T6, TiM) =
   qidError(QIL) .


*** now for one that actually works!
ceq preprocessPTimedTB(ME, M, D, VDS, DB, Q, T1, T6, TiM) =
    procParsedPTimedCommandTB(Q, MOD, TERM, BOUND,
                              CHEAPER, COSTLIMIT, SOLVEDTICKMODE)
if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
   /\ CHEAPER := ctComp(Q)
   /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D, VDS,
                                                 DB)
   /\  COSTLIMIT := solveBubbles(T6, MOD, B,
                                 getVars(getName(MOD),DB), DB)
   /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                       getVars(getName(MOD), DB), DB) .

 *** price-timed rewrites with term, timelimit
 *** this is essentially a slightly modified of
 *** preprocessTimedCommandTPCLCL
 op preprocessPTimedTL : ModuleExpression Module Bound
                         OpDeclSet Database Qid Term Term TickMode -> QidList .


 *** bad init term
 ceq preprocessPTimedTL(ME, M, D, VDS, DB, Q, T1, T4,
                         TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
          'does 'not 'parse. '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                       Tuple<Term|Module|OpDeclSet|Bound> .



  *** bad time limit
  ceq preprocessPTimedTL(ME, M, D, VDS, DB, Q, T1,
```

```
                                T4, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Time 'limit 'term
          'does 'not 'parse 'in 'module
          '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D,
                                                       VDS, DB)
         /\ not
            (solveBubbles(T4, MOD, B, getVars(getName(MOD), DB), DB) :: Term) .

*** bad tick mode
ceq preprocessPTimedTL(ME, M, D, VDS, DB, Q,T1, T4,
                       TiM) =
    ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
    printTickAmount(TiM)
            'does 'not 'parse 'in 'module
            '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
        if B := included('META-MODULE,
                         getImports(getTopModule(ME, DB)), DB)
           /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                          D, VDS, DB)
           /\  solveBubbles(T4, MOD, B,
                            getVars(getName(MOD), DB), DB) :: Term
           /\  not solveTickMode(TiM, MOD, B,
                                 getVars(getName(MOD), DB),
                                 DB) :: TickMode .

eq preprocessPTimedTL(ME, unitError(QIL), D, VDS, DB, Q,
                      T1, T4, TiM) =
   qidError(QIL) .


*** now for one that actually works!
ceq preprocessPTimedTL(ME, M, D, VDS, DB, Q, T1, T4, TiM) =
    procParsedPTimedCommandTL(Q, MOD, TERM, BOUND, COMP, LIMIT,
                              SOLVEDTICKMODE)
if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
   /\ COMP :=  commandToComp(Q)
   /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D, VDS,
                                                 DB)
   /\  LIMIT := solveBubbles(T4, MOD, B, getVars(getName(MOD), DB),
                             DB)
   /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                       getVars(getName(MOD), DB), DB) .

 *** price-timed rewrites with term
 *** this is essentially a slightly modified of
 *** preprocessTimedCommandTPCLCL
 op preprocessPTimedT : ModuleExpression Module Bound
                        OpDeclSet Database Qid Term TickMode -> QidList .


 *** bad init term
 ceq preprocessPTimedT(ME, M, D, VDS, DB, Q, T1, TiM) =
         ('\n '\r 'Error '\c 'in 'priced-timed 'command:  'Command/module/initterm
          'does 'not 'parse. '\o '\n)
      if B := included('META-MODULE,
                       getImports(getTopModule(ME, DB)), DB)
         /\  not solveBubblesRew(T1, M, B, D, VDS, DB) ::
                       Tuple<Term|Module|OpDeclSet|Bound> .

*** bad tick mode
ceq preprocessPTimedT(ME, M, D, VDS, DB, Q,T1, TiM) =
    ('\n '\r 'Error '\c 'in 'priced-timed 'command: 'Tick 'mode 'amount
    printTickAmount(TiM)
            'does 'not 'parse 'in 'module
            '\y eMetaPrettyPrint(getName(MOD)) '\o '\n)
        if B := included('META-MODULE,
                         getImports(getTopModule(ME, DB)), DB)
           /\  {TERM, MOD, ODS,  BOUND} := solveBubblesRew(T1, M, B,
                                                          D, VDS, DB)
           /\  not solveTickMode(TiM, MOD, B,
                                 getVars(getName(MOD), DB),
                                 DB) :: TickMode .

eq preprocessPTimedT(ME, unitError(QIL), D, VDS, DB, Q,
                     T1, TiM) =
   qidError(QIL) .


*** now for one that actually works!
ceq preprocessPTimedT(ME, M, D, VDS, DB, Q, T1, TiM) =
    procParsedPTimedCommandT(Q, MOD, TERM, BOUND,
                             SOLVEDTICKMODE)
if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
   /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T1, M, B, D, VDS,
                                                 DB)
   /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                       getVars(getName(MOD), DB), DB) .

*** THE ACTUAL CALLS aND RESULTS

*** procTimedSearch2 calles timedMetaSearch and does the job.
  *** In this first version, timedMetaSearch does all the module
```

```
    *** transformations. Therefore it can be slightly slow
    *** if we are looking for many solutions. However, that is a trivial
    *** improvement if needed.
  ***(
    op procTimedSearch2 : Module Term Term Condition Qid Bound Nat
                          ComparisonOp Term Bound TickMode -> QidList .
    *** procTimedSearch2(module, initterm, pattern, condition,
    ***                  arrowkind, depthOfRewrites, solNo, limit,
    ***                  noOfSolsSought, tickMode)

    eq procTimedSearch2(MOD, TERM, T, COND, Q, D, N, COMP, LIMIT,
                        BOUND, SOLVEDTICKMODE) =
         if
           timedMetaSearch(MOD, TERM, T, COND, Q, D, N, COMP,
                           LIMIT, SOLVEDTICKMODE)
           :: ResultTriple
         then
           ('\n  '\c 'Solution qid(string(N + 1, 10))
            '\n '\o eMetaPrettyPrint(MOD, getSubstitution(timedMetaSearch(MOD,
                        TERM, T, COND, Q, D, N, COMP, LIMIT, SOLVEDTICKMODE)))
            '\n
            (if N + 1 < BOUND
             then procTimedSearch2(MOD, TERM, T, COND, Q, D,
                                   N + 1, COMP, LIMIT, BOUND, SOLVEDTICKMODE)
             else nil fi)
            )
         else
         (if N == 0 then '\n '\c 'No 'solution '\o '\n
          else '\n   '\c 'No 'more 'solutions '\o '\n fi)
         fi .
)***

  *** PTM priced-timed search with time and cost limit
  *** Arguments:
  *** - Module
  *** - Initial state
  *** - Search pattern
  *** - Such that condition
  *** - '* '+ etc for now only '* is used
  *** - Bound, not used
  *** - lt, le
  *** - Time limit
  *** - Tick mode
  *** - Cheaper than if true
  *** - Cost limit
  *** Need sthing liek preprocessTimedCommandTPCLCL to squeeze all
  *** the bubbles out
  op procPricedTimedSearch : Module Term Term Condition Qid Bound Nat
                             ComparisonOp Term Bound TickMode Bool Term
                             -> QidList .

  op procTimedSearch2 : Module Term Term Condition Qid Bound Nat
                        ComparisonOp Term Bound TickMode -> QidList .

  ceq procPricedTimedSearch(MOD, TERM, T, COND, Q, D, N, COMP, LIMIT,
                            BOUND, SOLVEDTICKMODE, B, COSTLIMIT) =
      if
       R3? :: ResultTriple
      then
       ('\n  '\c 'Solution qid(string(N + 1, 10))
        '\n '\o eMetaPrettyPrint(MOD, getSubstitution(R3?))
        '\n
        (if N + 1 < BOUND
             then procPricedTimedSearch(MOD, TERM, T, COND, Q, D,
                                        N + 1, COMP, LIMIT, BOUND, SOLVEDTICKMODE,
                                        B, COSTLIMIT)
             else nil fi))
      else
       '\n '\c 'No 'more 'solutions '\o '\n
      fi
  *** here we should metareduce the initial term, and add
  *** a cost to both the initial and search terms
  if R3? :=
         pricedTimedSearch(MOD, TERM, T, COND,
                           Q, D, N, COMP,
                           LIMIT, SOLVEDTICKMODE,
                           B, getTerm(metaReduce(MOD, COSTLIMIT))) .

  op procPricedTimedSearch : Module Term Term Condition Qid Bound Nat
                             ComparisonOp Term Bound TickMode
                             -> QidList .
  ceq procPricedTimedSearch(MOD, TERM, T, COND, Q, D, N, COMP, LIMIT,
                            BOUND, SOLVEDTICKMODE) =
      if
       R3? :: ResultTriple
      then
       ('\n  '\c 'Solution qid(string(N + 1, 10))
        '\n '\o eMetaPrettyPrint(MOD, getSubstitution(R3?))
        '\n
        (if N + 1 < BOUND
             then procPricedTimedSearch(MOD, TERM, T, COND, Q, D,
                                        N + 1, COMP, LIMIT, BOUND, SOLVEDTICKMODE)
             else nil fi))
      else
       '\n '\c 'No 'more 'solutions '\o '\n
```

```
    fi
*** here we should metareduce the initial term, and add
*** a cost to both the initial and search terms
if R3? :=
       pricedTimedSearch(MOD, TERM, T,
                     COND,  Q, D, N, COMP,
                         LIMIT, SOLVEDTICKMODE) .


 *** ptsearch with time but no cost limit
 *** this one simply just reuses the parses required for
 *** an initial term, pattern and a time limit
 *** passes the result of the parse on to procPricedTimedSearch with no cost limit
 ceq procParsedTimedCommandTPL(Q, MOD, TERM, BOUND, T, COND, LIMIT, SOLVEDTICKMODE) =
        ('\n '\c 'Priced-timed 'search
               (if BOUND =/= unbounded    *** write [13] etc
                then ('\s ''[ qid(string(BOUND, 10)) '`] '\s )
                else nil fi) 'in '\o
         eMetaPrettyPrint(getName(MOD)) '\s '\n '\t
         eMetaPrettyPrint(MOD, TERM) qid("=>" + string(searchQid(Q))) '\s
         eMetaPrettyPrint(MOD, T) '\n
         '\c 'in 'time
         (if COMP == lt then '<
          else (if COMP == le then '<=
                else (if COMP == gt then '> else '>= fi) fi) fi)
         eMetaPrettyPrint(MOD, LIMIT) '\c
         'with 'no 'cost 'limit  '\c
        'and 'with 'mode
         printMode(SOLVEDTICKMODE, MOD)  ': '\n '\o
         *** Here comes the real call:
         procPricedTimedSearch(MOD,getTerm(metaReduce(MOD, TERM)),
                      getTerm(metaReduce(MOD, T)), COND,
                               if searchQid(Q) == '1 then
                                '+ else searchQid(Q) fi,
                               if searchQid(Q) == '1 then
                                1 else unbounded fi,
                               0, COMP, LIMIT, BOUND, SOLVEDTICKMODE))
      if
      COMP :=  commandToComp(Q) /\
      ((find(string(Q), "ptsearch", 0) =/= notFound) and
         (find(string(Q), "in`time", 0) =/= notFound) and
         --- added to differentiate from no cost limit
         (find(string(Q), "with`no`cost`limit", 0) =/= notFound)) .


*** PTM find cheapest with time limit
*** maybe some more preprocessing is needed
*** Arguments:
*** - Module
*** - Initial state
*** - Search pattern
*** - Such that condition
*** - '* '+ etc for now only '* is used
*** - Bound, not used
*** - lt, le
*** - Time limit
*** - Tick mode
op procFindCheapest : Module Term Term Condition Qid Bound
                      ComparisonOp Term TickMode -> QidList .

ceq procFindCheapest(MOD, TERM, T, COND, Q, D, COMP, LIMIT,
                     SOLVEDTICKMODE) =
    if
     R3? :: ResultTriple
    then
     ('\n  '\c 'Solution
      '\n '\o eMetaPrettyPrint(MOD, getSubstitution(R3?))
      '\n)
    else
     '\n '\c 'No 'solution '\o '\n
    fi
*** here we should metareduce the initial term, and add
*** a cost to both the initial and search terms
if R3? :=
      if (find(string(Q), "binary", 0) == notFound) then
       findCheapest(MOD, getTerm(metaReduce(MOD, TERM)),
                    getTerm(metaReduce(MOD, T)),
                    COND, Q, D, COMP,
                    LIMIT, SOLVEDTICKMODE)
       else
        *** binary
        findCheapestBin(MOD, getTerm(metaReduce(MOD, TERM)),
                    getTerm(metaReduce(MOD, T)),
                        COND, Q, D, COMP,
                        LIMIT, SOLVEDTICKMODE)
       fi .


op procUtFindCheapest : Module Term Term Condition Qid Bound
                        -> QidList .

ceq procUtFindCheapest(MOD, TERM, T, COND, Q, D) =
    if
     R3? :: ResultTriple
    then
```

```
     ('\n  '\c 'Solution
      '\n '\o eMetaPrettyPrint(MOD, getSubstitution(R3?))
      '\n)
    else
     '\n '\c 'No 'solution '\o '\n
    fi
*** here we should metareduce the initial term, and add
*** a cost to both the initial and search terms
if R3? :=   utFindCheapest(MOD, getTerm(metaReduce(MOD, TERM)),
                      getTerm(metaReduce(MOD, T)),
                           COND, Q, D) .

op procPricedSearch : Module Term Term Condition Qid Bound
                      Bool Term -> QidList .

ceq procPricedSearch(MOD, TERM, T, COND, Q, D, CHEAPER, COSTLIMIT) =
    if
     R3? :: ResultTriple
    then
     ('\n  '\c 'Solution
      '\n '\o eMetaPrettyPrint(MOD, getSubstitution(R3?))
      '\n)
    else
     '\n '\c 'No 'solution '\o '\n
    fi
*** here we should metareduce the initial term, and add
*** a cost to both the initial and search terms
if R3? :=   pricedSearch(MOD, TERM,
                      T,
                         COND, Q, D, CHEAPER, COSTLIMIT) .

*** find cheapest with time limit
*** this one simply just reuses the parses required for
*** an initial term, pattern and a time limit
ceq procParsedTimedCommandTPL(Q, MOD, TERM, BOUND, T, COND, LIMIT,
                              SOLVEDTICKMODE) =

    ('\n '\c
    if (find(string(Q), "binary", 0) == notFound) then
     'Find
    else
     'Binary 'find
    fi
    'cheapest
    (if BOUND =/= unbounded    *** write [13] etc
     then
      ('\s '`[ qid(string(BOUND, 10)) '`] '\s )
     else nil fi)
     'in '\o eMetaPrettyPrint(getName(MOD)) '\c '\s '\n '\t
     eMetaPrettyPrint(MOD, TERM)
     qid("=>" + string(searchQid(Q))) '\s
     eMetaPrettyPrint(MOD, T) '\n '\c
     'in 'time commandToCompSymb(Q)
     eMetaPrettyPrint(MOD, LIMIT) 'and 'with 'mode
     printMode(SOLVEDTICKMODE, MOD)   ': '\n '\o
     *** Here comes the real call:
     procFindCheapest(MOD, TERM, T, COND,
                      if searchQid(Q) == '1 then
                       '+
                      else
                       searchQid(Q)
                      fi,
                      if searchQid(Q) == '1 then
                       1
                      else
                       unbounded
                      fi,
                      commandToComp(Q), LIMIT, SOLVEDTICKMODE))
if (find(string(Q), "find`cheapest_=>*_in`time`<", 0) =/= notFound) .
---(Q == 'find`cheapest_=>*_in`time`<=_.)
---   or (Q == 'find`cheapest_=>*_in`time`<_.) .

op procParsedPricedTPB : Qid Module Term Bound Term Condition
                         Bool Term -> QidList .
ceq procParsedPricedTPB(Q, MOD, TERM, BOUND, T, COND,
                        CHEAPER, COSTLIMIT) =

    ('\n '\c 'Priced 'Search
    (if BOUND =/= unbounded    *** write [13] etc
     then
      ('\s '`[ qid(string(BOUND, 10)) '`] '\s )
     else nil fi)
     'in '\o eMetaPrettyPrint(getName(MOD)) '\c '\s '\n '\t
     eMetaPrettyPrint(MOD, TERM)
     qid("=>" + string(searchQid(Q))) '\s
     eMetaPrettyPrint(MOD, T) '\n '\c
     *** Here comes the real call:
     procPricedSearch(MOD, getTerm(metaReduce(MOD, TERM)),
                   getTerm(metaReduce(MOD, T)), COND,
                      if searchQid(Q) == '1 then
                       '+
                        else
                        searchQid(Q)
                        fi,
                        if searchQid(Q) == '1 then
```

```
                        1
                       else
                        unbounded
                       fi, CHEAPER, COSTLIMIT))
if (find(string(Q), "psearch", 0) =/= notFound) .

op procParsedPTimedCommandTP : Qid Module Term Bound Term
                               Condition TickMode -> QidList .

*** Find earliest: no cost limit.
ceq procParsedPTimedCommandTP('priced`find`earliest_=>*_with`no`cost`limit., MOD, TERM, BOUND,
                              T, COND, SOLVEDTICKMODE) =
      ('\n '\c 'Priced 'find 'earliest '\s '\o
       eMetaPrettyPrint(MOD, T) '\c  'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c 'such 'that '\s '\n '\t
       '\o eMetaPrettyPrint(MOD, TERM) '=>* '\s
       eMetaPrettyPrint(MOD, T) '\n
       '\y 'with 'no 'cost 'limit
       '\c 'with 'mode
       printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
       *** Here comes the real call:
       (if RT :: ResultTriple then
          ('\c '\n 'Result: '\o '\t
           eMetaPrettyPrint(MOD, metaMoveCost(getTerm(RT))) '\o '\n )
        else (if RT == failure then
                ('\c '\n 'Result: 'state 'not 'reachable. '\o '\n)
              else ('\n '\r 'Error: 'something 'went 'wrong. '\o '\n)
              fi)
        fi)
      )
    if RT := findEarliest(pricifyMod(MOD),
                          pricifyInit(getTerm(metaReduce(MOD, TERM))),
                  pricifyPattern(getTerm(metaReduce(MOD, T))),
                          COND, SOLVEDTICKMODE) .

*** find cheapest with no time limit
ceq procParsedPTimedCommandTP(Q, MOD, TERM, BOUND, T, COND,
                              SOLVEDTICKMODE) =

    ('\n '\c
    if (find(string(Q), "binary", 0) == notFound) then
     'Find
    else
     'Binary 'find
    fi
    'cheapest
    (if BOUND =/= unbounded    *** write [13] etc
     then
      ('\s '`[ qid(string(BOUND, 10)) '`] '\s )
     else nil fi)
     'in '\o eMetaPrettyPrint(getName(MOD)) '\c '\s '\n '\t
     eMetaPrettyPrint(MOD, TERM)
     qid("=>" + string(searchQid(Q))) '\s
     eMetaPrettyPrint(MOD, T) '\n '\c
     'with 'no 'time 'limit 'time 'and 'with 'mode
     printMode(SOLVEDTICKMODE, MOD)   ': '\n '\o
     *** Here comes the real call:
     procFindCheapest(MOD, TERM, T, COND,
                      if searchQid(Q) == '1 then
                       '+
                      else
                       searchQid(Q)
                      fi,
                      if searchQid(Q) == '1 then
                       1
                      else
                       unbounded
                      fi,
                      ge, 'zero.Time, SOLVEDTICKMODE))
if (find(string(Q), "with`no", 0) =/= notFound) and
   (find(string(Q), "find`cheapest", 0) =/= notFound) .

*** untimed find cheapest, could make a different parse path
*** considering it's not ptimed
ceq procParsedPTimedCommandTP(Q, MOD, TERM, BOUND, T, COND,
                              SOLVEDTICKMODE) =

    ('\n '\c 'Un-timed 'Find 'cheapest
    (if BOUND =/= unbounded    *** write [13] etc
     then
      ('\s '`[ qid(string(BOUND, 10)) '`] '\s )
     else nil fi)
     'in '\o eMetaPrettyPrint(getName(MOD)) '\c '\s '\n '\t
     eMetaPrettyPrint(MOD, TERM)
     qid("=>" + string(searchQid(Q))) '\s
     eMetaPrettyPrint(MOD, T) '\n '\c
     *** Here comes the real call:
     procUtFindCheapest(MOD, TERM, T, COND,
                        if searchQid(Q) == '1 then
                         '+
                        else
                        searchQid(Q)
                        fi,
                        if searchQid(Q) == '1 then
                         1
```

```
                       else
                        unbounded
                       fi))
if (find(string(Q), "ut`find`cheapest", 0) =/= notFound) .



*** ptsearch with no limits
ceq procParsedPTimedCommandTP(Q, MOD, TERM, BOUND, T, COND,
                              SOLVEDTICKMODE) =

      ('\n '\c 'Priced-timed 'search
             (if BOUND =/= unbounded    *** write [13] etc
              then ('\s ''[ qid(string(BOUND, 10)) '`] '\s )
              else nil fi) 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\s '\n '\t
       eMetaPrettyPrint(MOD, TERM) qid("=>" + string(searchQid(Q))) '\s
       eMetaPrettyPrint(MOD, T) '\n
       '\c 'with 'no 'time 'or 'cost 'limit
      'and 'with 'mode
       printMode(SOLVEDTICKMODE, MOD)  ': '\n '\o
       *** Here comes the real call:
       procPricedTimedSearch(MOD,
                             getTerm(metaReduce(MOD, TERM)),
                getTerm(metaReduce(MOD, T)), COND,
                             if searchQid(Q) == '1 then
                              '+ else searchQid(Q) fi,
                             if searchQid(Q) == '1 then
                              1 else unbounded fi,
                             0, ge, 'zero.Time, BOUND, SOLVEDTICKMODE,
                             true, 'infcost.CostInf))
if ((find(string(Q), "with`no`limits", 0) =/= notFound) and
   (find(string(Q), "ptsearch", 0) =/= notFound)) .

*** Process priced timed rewrites with time and cost limit
op procParsedPTimedCommandTLB : Qid Module Term Bound
                                ComparisonOp Term
                                Bool Term
                                TickMode -> QidList .

*** priced timed fair rewrites with time and cost limit
ceq procParsedPTimedCommandTLB(Q, MOD, TERM, BOUND, COMP, LIMIT,
                               CHEAPER, COSTLIMIT, SOLVEDTICKMODE) =
    if RP :: ResultPair then
     ('\n '\c 'Priced-timed if FAIR then 'fair 'rewrite
                            else 'rewrite fi
      (if BOUND =/= unbounded    *** write [13] etc
       then ('\s ''[ qid(string(BOUND, 10)) '`] )
       else nil fi) '\o '\s '\s
        eMetaPrettyPrint(MOD, TERM) '\c  'in  '\o
        eMetaPrettyPrint(getName(MOD)) '\c 'with 'mode
        printMode(SOLVEDTICKMODE, MOD)
        'in 'time
        (if COMP == lt then
          '<
         else
          '<=
         fi) eMetaPrettyPrint(MOD, LIMIT) '\c
       'with 'cost
       (if CHEAPER then
         '<
        else
         '<=
        fi)
        eMetaPrettyPrint(MOD, COSTLIMIT)
        '\c '\n 'Result '\o
        eMetaPrettyPrint(leastSort(MOD, metaMoveCost(getTerm(RP)))) '\c ': '\o '\n '\s '\s
        eMetaPrettyPrint(MOD, metaMoveCost(getTerm(RP))) '\n)
       else ('\r 'Error 'in 'timed 'rewrite.  '\o '\n)
       fi
  if *** maybe move this to the preparser
     ((find(string(Q), "ptfrew_in`time", 0) =/= notFound) or
     (find(string(Q), "ptrew_in`time", 0) =/= notFound))
     /\ FAIR := (find(string(Q), "ptfrew", 0) =/= notFound)
     /\ RP :=
        if FAIR then
         pricedTimedMetaFRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND, 1,
                                 COMP, LIMIT, SOLVEDTICKMODE, CHEAPER, COSTLIMIT)
         else
         pricedTimedMetaRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND,
                                COMP, LIMIT, SOLVEDTICKMODE, CHEAPER, COSTLIMIT)
         fi .


*** now for pt(f)rew with no time limit
op procParsedPTimedCommandTB : Qid Module Term Bound
                               Bool Term TickMode -> QidList .

  *** pt(f)rew with no time limit
  ceq procParsedPTimedCommandTB(Q, MOD, TERM, BOUND,
                                CHEAPER, COSTLIMIT, SOLVEDTICKMODE) =
      if RP :: ResultPair then
       ('\n '\c 'Priced-timed if FAIR then 'fair 'rewrite
                              else 'rewrite fi
```

```
        (if BOUND =/= unbounded    *** write [13] etc
         then ('\s ''[ qid(string(BOUND, 10)) ''] )
         else nil fi) '\o '\s '\s
          eMetaPrettyPrint(MOD, TERM) '\c  'in  '\o
          eMetaPrettyPrint(getName(MOD)) '\c 'with 'mode
          printMode(SOLVEDTICKMODE, MOD)
          'with 'no 'time 'limit 'and
         'with 'cost
         (if CHEAPER then
           '<
          else
           '<=
         fi)
         eMetaPrettyPrint(MOD, COSTLIMIT)
         '\c '\n 'Result '\o
         eMetaPrettyPrint(leastSort(MOD, metaMoveCost(getTerm(RP)))) '\c ': '\o '\n '\s '\s
         eMetaPrettyPrint(MOD, metaMoveCost(getTerm(RP))) '\n)
        else ('\r 'Error 'in 'timed 'rewrite.  '\o '\n)
        fi
    if *** maybe move this to the preparser
       ((find(string(Q), "ptfrew_with`no", 0) =/= notFound) or
       (find(string(Q), "ptrew_with`no", 0) =/= notFound))
       /\ FAIR := (find(string(Q), "ptfrew", 0) =/= notFound)
       /\ RP :=
          if FAIR then
           pricedTimedMetaFRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND, 1,
                                   ge, 'zero.Time, SOLVEDTICKMODE, CHEAPER, COSTLIMIT)
           else
           pricedTimedMetaRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND,
                                  ge, 'zero.Time, SOLVEDTICKMODE, CHEAPER, COSTLIMIT)
        fi .

*** Process priced timed rewrites with time limit
op procParsedPTimedCommandTL : Qid Module Term Bound
                               ComparisonOp Term
                               TickMode -> QidList .

*** priced timed fair rewrites with time limit
ceq procParsedPTimedCommandTL(Q, MOD, TERM, BOUND, COMP, LIMIT,
                              SOLVEDTICKMODE) =
    if RP :: ResultPair then
     ('\n '\c 'Priced-timed if FAIR then 'fair 'rewrite
                            else 'rewrite fi
      (if BOUND =/= unbounded    *** write [13] etc
       then ('\s ''[ qid(string(BOUND, 10)) ''] )
       else nil fi) '\o '\s '\s
        eMetaPrettyPrint(MOD, TERM) '\c  'in  '\o
        eMetaPrettyPrint(getName(MOD)) '\c 'with 'mode
        printMode(SOLVEDTICKMODE, MOD)
        'in 'time
        (if COMP == lt then
          '<
         else
          '<=
         fi) eMetaPrettyPrint(MOD, LIMIT) '\c
       'with 'no  'cost 'limit
        '\c '\n 'Result '\o
        eMetaPrettyPrint(leastSort(MOD, metaMoveCost(getTerm(RP)))) '\c ': '\o '\n '\s '\s
        eMetaPrettyPrint(MOD, metaMoveCost(getTerm(RP))) '\n)
       else ('\r 'Error 'in 'timed 'rewrite.  '\o '\n)
       fi
  if *** maybe move this to the preparser
     ((find(string(Q), "ptfrew_in`time", 0) =/= notFound) or
     (find(string(Q), "ptrew_in`time", 0) =/= notFound))

     /\ FAIR := (find(string(Q), "ptfrew", 0) =/= notFound)
     /\ RP :=
        if FAIR then
         pricedTimedMetaFRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND, 1,
                                 COMP, LIMIT, SOLVEDTICKMODE)
         else
         pricedTimedMetaRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND,
                                COMP, LIMIT, SOLVEDTICKMODE)
         fi .

*** now for pt(f)rew with no limits
op procParsedPTimedCommandT : Qid Module Term Bound TickMode -> QidList .

  *** pt(f)rew with no time limit
  ceq procParsedPTimedCommandT(Q, MOD, TERM, BOUND, SOLVEDTICKMODE) =
      if RP :: ResultPair then
       ('\n '\c 'Priced-timed if FAIR then 'fair 'rewrite
                              else 'rewrite fi
        (if BOUND =/= unbounded    *** write [13] etc
         then ('\s ''[ qid(string(BOUND, 10)) ''] )
         else nil fi) '\o '\s '\s
          eMetaPrettyPrint(MOD, TERM) '\c  'in  '\o
          eMetaPrettyPrint(getName(MOD)) '\c 'with 'mode
          printMode(SOLVEDTICKMODE, MOD)
          'with 'no 'limits
          '\c '\n 'Result '\o

          eMetaPrettyPrint(leastSort(MOD, metaMoveCost(getTerm(RP)))) '\c ': '\o '\n '\s '\s
          eMetaPrettyPrint(MOD, metaMoveCost(getTerm(RP))) '\n)
```

```
        else ('\r 'Error 'in 'timed 'rewrite.  '\o '\n)
        fi
    if *** maybe move this to the preparser
       ((find(string(Q), "ptfrew_with`no", 0) =/= notFound) or
       (find(string(Q), "ptrew_with`no", 0) =/= notFound))
       /\ FAIR := (find(string(Q), "ptfrew", 0) =/= notFound)
       /\ RP :=
          if FAIR then
           pricedTimedMetaFRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND, 1,
                                   ge, 'zero.Time, SOLVEDTICKMODE)
           else
           pricedTimedMetaRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND,
                                  ge, 'zero.Time, SOLVEDTICKMODE)
        fi .


  *** untimed priced rewrite
  ceq procParsedPTimedCommandTB(Q, MOD, TERM, BOUND,
                                  CHEAPER, COSTLIMIT, SOLVEDTICKMODE) =
      if RP :: ResultPair then
       ('\n '\c 'Priced-timed if FAIR then 'fair 'rewrite
                             else 'rewrite fi
        (if BOUND =/= unbounded    *** write [13] etc
         then ('\s ''[ qid(string(BOUND, 10)) '`] )
         else nil fi) '\o '\s '\s
          eMetaPrettyPrint(MOD, TERM) '\c  'with  '\o
          'cost
         (if CHEAPER then
           '<
          else
           '<=
          fi)
          eMetaPrettyPrint(MOD, COSTLIMIT)
          '\c '\n 'Result '\o
          eMetaPrettyPrint(getType(RP)) '\c ': '\o '\n '\s '\s
          eMetaPrettyPrint(MOD, getTerm(RP)) '\n)
         else ('\r 'Error 'in 'priced 'rewrite.  '\o '\n)
         fi
    if *** maybe move this to the preparser
       ((find(string(Q), "pfrew", 0) =/= notFound) or
       (find(string(Q), "prew", 0) =/= notFound))
       /\ FAIR := (find(string(Q), "pfrew", 0) =/= notFound)
       /\ RP :=
          if FAIR then
           pricedMetaFRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND, 1, CHEAPER,
                              COSTLIMIT)
           else
           pricedMetaRewrite(MOD, getTerm(metaReduce(MOD, TERM)), BOUND, CHEAPER, COSTLIMIT)
        fi .



*** 'ptfrew_in`time`<=_cheaper`than_. or
*** 'ptfrew_in`time`<=_cheaper`than`or`eq_. or
*** 'ptfrew_in`time`<_cheaper`than_. or
*** 'ptfrew_in`time`<_cheaper`than`or`eq_. or
*** 'ptfrew_with`no`time`limit`cheaper`than_. or
*** 'ptfrew_with`no`time`limit`cheaper`than`or`eq_. or
*** 'ptrew_in`time`<=_cheaper`than_. or
*** 'ptrew_in`time`<=_cheaper`than`or`eq_. or
*** 'ptrew_in`time`<_cheaper`than_. or
*** 'ptrew_in`time`<_cheaper`than`or`eq_. or
*** 'ptrew_with`no`time`limit`cheaper`than_. or
*** 'ptrew_with`no`time`limit`cheaper`than`or`eq_. .
*** 'ptfrew_in`time`<=_cheaper`than_.
*** 'ptfrew_in`time`<=_cheaper`than`or`eq_.
*** 'ptfrew_in`time`<_cheaper`than_.
*** 'ptfrew_in`time`<_cheaper`than`or`eq_.

  *** ptsearch with time and cost limit
  op procParsedPTimedCommandTPLB : Qid Module Term Bound Term
                                   Condition ComparisonOp Term
                                   Bool Term
                                   TickMode -> QidList .

  ceq procParsedPTimedCommandTPLB(Q, MOD, TERM, BOUND, T, COND,
                                  COMP, LIMIT, CHEAPER, COSTLIMIT, SOLVEDTICKMODE) =

       ('\n '\c 'Priced-timed 'search
             (if BOUND =/= unbounded    *** write [13] etc
              then ('\s ''[ qid(string(BOUND, 10)) '`] '\s )
              else nil fi) 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\s '\n '\t
       eMetaPrettyPrint(MOD, TERM) qid("=>" + string(searchQid(Q))) '\s
       eMetaPrettyPrint(MOD, T) '\n
       '\c 'in 'time
       (if COMP == lt then '<
        else (if COMP == le then '<=
              else (if COMP == gt then '> else '>= fi) fi) fi)
       eMetaPrettyPrint(MOD, LIMIT) '\c
       'with 'cost
             (if CHEAPER then
               '<
             else
               '<=
```

```
             fi)
        eMetaPrettyPrint(MOD, COSTLIMIT) '\c
       'and 'with 'mode
        printMode(SOLVEDTICKMODE, MOD)  ': '\n '\o
        *** Here comes the real call:
        procPricedTimedSearch(MOD,
                              getTerm(metaReduce(MOD, TERM)),
                         getTerm(metaReduce(MOD, T)), COND,
                              if searchQid(Q) == '1 then
                               '+ else searchQid(Q) fi,
                              if searchQid(Q) == '1 then
                               1 else unbounded fi,
                              0, COMP, LIMIT, BOUND, SOLVEDTICKMODE, CHEAPER, COSTLIMIT))
     if ((find(string(Q), "ptsearch", 0) =/= notFound) and
        (find(string(Q), "in`time", 0) =/= notFound)) and
        --- added to differentiate from no cost limit
        (find(string(Q), "with`cost", 0) =/= notFound) .

  *** ptsearch with no time limit, but cost limit
  op procParsedPTimedCommandTPB : Qid Module Term Bound Term
                                  Condition Bool Term
                                  TickMode -> QidList .

  ceq procParsedPTimedCommandTPB(Q, MOD, TERM, BOUND, T, COND,
                                 CHEAPER, COSTLIMIT, SOLVEDTICKMODE) =

             ('\n '\c 'Priced-timed 'search
                    (if BOUND =/= unbounded    *** write [13] etc
                     then ('\s ''[ qid(string(BOUND, 10)) '`] '\s )
                     else nil fi) 'in '\o
              eMetaPrettyPrint(getName(MOD)) '\s '\n '\t
              eMetaPrettyPrint(MOD, TERM) qid("=>" + string(searchQid(Q))) '\s
              eMetaPrettyPrint(MOD, T) '\n
              '\c 'with 'no 'time 'limit
              'with 'cost
                    (if CHEAPER then
                      '<
                    else
                      '<=
                    fi)
              eMetaPrettyPrint(MOD, COSTLIMIT) '\c
             'and 'with 'mode
              printMode(SOLVEDTICKMODE, MOD)  ': '\n '\o
              *** Here comes the real call:
              procPricedTimedSearch(MOD,
                            getTerm(metaReduce(MOD, TERM)),
                            getTerm(metaReduce(MOD, T)), COND,
                                     if searchQid(Q) == '1 then
                                      '+ else searchQid(Q) fi,
                                     if searchQid(Q) == '1 then
                                      1 else unbounded fi,
                                     0, ge, 'zero.Time, BOUND, SOLVEDTICKMODE, CHEAPER, COSTLIMIT))
           if ((find(string(Q), "ptsearch", 0) =/= notFound) and
              (find(string(Q), "with`no`time", 0) =/= notFound)) .

  *** Find earliest: with cost limit.
  ceq procParsedPTimedCommandTPB(Q, MOD, TERM, BOUND, T, COND,
                                 CHEAPER, COSTLIMIT, SOLVEDTICKMODE) =
      ('\n '\c 'Priced 'find 'earliest '\s '\o
         eMetaPrettyPrint(MOD, T) '\c  'in '\o
         eMetaPrettyPrint(getName(MOD)) '\c 'such 'that '\s '\n '\t
         '\o eMetaPrettyPrint(MOD, TERM) '=>* '\s
         eMetaPrettyPrint(MOD, T) '\n
         '\y 'with 'cost
         (if CHEAPER then
             '<
          else
             '<=
          fi)
              eMetaPrettyPrint(MOD, COSTLIMIT) '\c
         '\c 'with 'mode
         printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
         *** Here comes the real call:
         (if RT :: ResultTriple then
            ('\c '\n 'Result: '\o '\t
             eMetaPrettyPrint(MOD, metaMoveCost(getTerm(RT))) '\o '\n )
          else (if RT == failure then
                  ('\c '\n 'Result: 'state 'not 'reachable. '\o '\n)
                else ('\n '\r 'Error: 'something 'went 'wrong. '\o '\n)
                fi)
          fi)
        )
      if RT := findEarliest(costLimitMod(pricifyMod(MOD),CHEAPER,COSTLIMIT),
                            pricifyInit(getTerm(metaReduce(MOD, TERM))),
                     pricifyPattern(getTerm(metaReduce(MOD, T))),
                            COND, SOLVEDTICKMODE)
         /\
         ((find(string(Q), "find`earliest", 0) =/= notFound) and
         (find(string(Q), "with`cost`"", 0) =/= notFound)) .


*** model checking commands
*** these are mainly unchanged from Real-Time Maude
*** all commands hav are prefixed with a p for priced
*** in addition for the last step the module is pricified,
```

```
*** initial term gets a with cost free added and any search pattern has a cost variable added
*** MOD ->
*** pricifyMod(MOD),
*** TERM ->
*** makePriced(getTerm(metaReduce(MOD, TERM)), getTerm(metaReduce(MOD,'free.Cost))),
*** T-> (?)
*** makePriced(getTerm(metaReduce(MOD,T)),newCostVar(T,'TOTAL_COST_INCURRED)),

*** All the preparsing can safely be done by the Real-Time Maude function

ceq procTimedCommand(Q[T, T', T''], ME, DB, TiM) =
        (if compiledModule(ME, DB)
         then preprocessTimedCommandTPP(ME, getFlatModule(ME, DB), unbounded,
                                        getVars(ME, DB), DB, Q, T, T', T'',TiM)
         else preprocessTimedCommandTPP(modExp(evalModExp(ME, DB)),
                                    getFlatModule(modExp(evalModExp(ME, DB)),
                                                  database(evalModExp(ME, DB))),
                                    unbounded,
                                    getVars(modExp(evalModExp(ME, DB)),
                                            database(evalModExp(ME, DB))),
                                    database(evalModExp(ME, DB)),
                                    Q, T, T', T'', TiM)
          fi)
      if (Q == 'pcheck_|=_until_with`no`time`limit`.) or
         (Q == 'pcheck_|=_untilStable_with`no`time`limit`.) .


  ceq procTimedCommand(Q[T, T', T'', T'''], ME, DB, TiM) =
          (if compiledModule(ME, DB)
           then preprocessTimedCommandTPPL(ME, getFlatModule(ME, DB), unbounded,
                                           getVars(ME, DB), DB, Q, T, T', T'',
                                           T''', TiM)
           else preprocessTimedCommandTPPL(modExp(evalModExp(ME, DB)),
                                     getFlatModule(modExp(evalModExp(ME, DB)),
                                                   database(evalModExp(ME, DB))),
                                     unbounded,
                                     getVars(modExp(evalModExp(ME, DB)),
                                             database(evalModExp(ME, DB))),
                                     database(evalModExp(ME, DB)),
                                     Q, T, T', T'', T''', TiM)
            fi)
        if (Q == 'pcheck_|=_until_in`time`<_.)         or
           (Q == 'pcheck_|=_until_in`time`<=_.)        or
           (Q == 'pcheck_|=_untilStable_in`time`<_.)   or
           (Q == 'pcheck_|=_untilStable_in`time`<=_.) .


  ceq procTimedCommand(Q[T, T'], ME, DB, TiM) =
        (if compiledModule(ME, DB)
         then preprocessTimedCommandTP(ME, getFlatModule(ME, DB), unbounded,
                                       getVars(ME, DB), DB, Q, T, T', TiM)
         else preprocessTimedCommandTP(modExp(evalModExp(ME, DB)),
                                    getFlatModule(modExp(evalModExp(ME, DB)),
                                                  database(evalModExp(ME, DB))),
                                    unbounded,
                                    getVars(modExp(evalModExp(ME, DB)),
                                            database(evalModExp(ME, DB))),
                                    database(evalModExp(ME, DB)), Q, T, T', TiM)
          fi)
      if (Q == 'pmc_|=u_.)                                 or
         (Q == 'pmc_|=t_with`no`time`limit`.) .

ceq procPriceTimedCommand(Q[TL], ME, DB, TiM) = procTimedCommand(Q[TL], ME, DB, TiM)
if Q == 'pcheck_|=`<>_with`no`time`limit`. or
   Q == 'pcheck_|=`<>_in`time`<_. or
   Q == 'pcheck_|=`<>_in`time`<=_. or
   Q == 'pcheck_|=_until_with`no`time`limit`. or
   Q == 'pcheck_|=_until_in`time`<_. or
   Q == 'pcheck_|=_until_in`time`<=_. or
   Q == 'pcheck_|=_untilStable_with`no`time`limit`. or
   Q == 'pcheck_|=_untilStable_in`time`<_. or
   Q == 'pcheck_|=_untilStable_in`time`<=_. or
   Q == 'pmc_|=u_. or
   Q == 'pmc_|=t_with`no`time`limit`. or
   Q == 'pmc_|=t_in`time`<_.  or
   Q == 'pmc_|=t_in`time`<=_. .


  *** Homemade diamond check:
  *** First, no time limit
  ceq procParsedTimedCommandTP('pcheck_|=`<>_with`no`time`limit`., MOD,
                               TERM, BOUND,
                               T, COND, SOLVEDTICKMODE) =
        ('\n '\c 'Check '\s '\o eMetaPrettyPrint(MOD, TERM)
         '\c '|= '\y '<> '\o '\s
         eMetaPrettyPrint(MOD, T) '\c 'in '\o
         eMetaPrettyPrint(getName(MOD)) '\c
         'with 'no 'time 'limit 'with 'mode
         printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
         *** Here comes the real call:
         (if TS :: Term then
            ('\c '\n 'Result: 'the 'property 'does 'not 'hold.
                 'Counterexample:  '\n '\o '\t
             eMetaPrettyPrint(MOD, TS) '\o '\n )
          else (if TS == noterm then
```

```
               ('\c '\n 'Result: 'the 'property 'holds. '\o '\n)
             else ('\r '\n 'Error: 'something 'went 'wrong. '\o '\n)
             fi)
        fi)
      )
    if TS := timedDiamond(pricifyProperties(pricifyMod(MOD)),
                 pricifyInit(getTerm(metaReduce(MOD, TERM))),
                         T,
                         COND, true, SOLVEDTICKMODE) .

*** Now with time constraints:
ceq procParsedTimedCommandTPL(Q, MOD,  TERM, BOUND,
                              T, COND,  LIMIT, SOLVEDTICKMODE) =
      ('\n '\c 'Check '\s '\o eMetaPrettyPrint(MOD, TERM)  '\c '|=
       '\y '<> '\o '\s
       eMetaPrettyPrint(MOD, T) '\c 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c
       'in 'time commandToCompSymb(Q)
       eMetaPrettyPrint(MOD, LIMIT) '\c
       'with 'mode
       printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
       *** Here comes the real call:
       (if TS :: Term then
          ('\c '\n 'Result: 'the 'property 'does 'not 'hold.
               'Counterexample: '\n '\o '\t
           eMetaPrettyPrint(MOD, TS) '\o '\n )
        else (if TS == noterm then
                  ('\c '\n 'Result: 'The 'property 'holds. '\o '\n)
             else ('\r '\n 'Error: 'something 'went 'wrong. '\o '\n)
             fi)
        fi)
      )
    if (Q == 'pcheck_|='<>_in'time'<_.) or
       (Q == 'pcheck_|='<>_in'time'<=_.)
       /\ TS := timedDiamond(pricifyProperties(pricifyMod(MOD)),
                             pricifyInit(getTerm(metaReduce(MOD, TERM))),
                             T,
                             COND, true, commandToComp(Q),
                             LIMIT, SOLVEDTICKMODE) .

*** Homemade "until". This also exists with an untimed version which
*** I do not provide here!

*** First without time constraints:
ceq procParsedTimedCommandTPP('pcheck_|=_until_with'no'time'limit'., MOD,
                              TERM, BOUND, T, COND,
                              T', COND',  SOLVEDTICKMODE) =
      ('\n '\c 'Check '\s '\o eMetaPrettyPrint(MOD, TERM)
      '\c '|= '\s '\o eMetaPrettyPrint(MOD, T) '\y 'until '\o '\s
       eMetaPrettyPrint(MOD, T') '\c 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c
       'with 'no 'time 'limit 'with 'mode
       printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
       *** Here comes the real call:
       (if TS :: Term then
          ('\c '\n 'Result: 'the 'property 'does 'not 'hold.
               'Counterexample:  '\n '\o '\t
           eMetaPrettyPrint(MOD, TS) '\o '\n )
        else (if TS == noterm then
                  ('\c '\n 'Result: 'The 'property 'holds. '\o '\n)
             else ('\r '\n 'Error: 'something 'went 'wrong. '\o '\n)
             fi)
        fi)
      )
    if TS := timedUntil(pricifyProperties(pricifyMod(MOD)),
                        pricifyInit(getTerm(metaReduce(MOD, TERM))),
                        T,
                        COND, true, T', COND',
                        true, SOLVEDTICKMODE) .

*** Now with time constraints:
ceq procParsedTimedCommandTPPL(Q, MOD,  TERM, BOUND, T, COND,
                               T', COND',  LIMIT, SOLVEDTICKMODE) =
      ('\n '\c 'Check '\s '\o eMetaPrettyPrint(MOD, TERM)
      '\c '|= '\s '\o eMetaPrettyPrint(MOD, T) '\y 'until '\o '\s
       eMetaPrettyPrint(MOD, T') '\c 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c
       'in 'time commandToCompSymb(Q)
       eMetaPrettyPrint(MOD, LIMIT) '\c
       'with 'mode
       printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
       *** Here comes the real call:
       (if TS :: Term then
          ('\c '\n 'Result: 'the 'property 'does 'not 'hold.
               'Counterexample: '\n '\o '\t
           eMetaPrettyPrint(MOD, TS) '\o '\n )
        else (if TS == noterm then
                  ('\c '\n 'Result: 'The 'property 'holds. '\o '\n)
             else ('\r '\n 'Error: 'something 'went 'wrong. '\o '\n)
             fi)
        fi)
      )
    if (Q == 'pcheck_|=_until_in'time'<_.) or
       (Q == 'pcheck_|=_until_in'time'<=_.)
       /\ TS := timedUntil(pricifyProperties(pricifyMod(MOD)),
```

```
                           pricifyInit(getTerm(metaReduce(MOD, TERM))),
                           T,
                           COND,  true, T', COND',
                           true, commandToComp(Q),  LIMIT, SOLVEDTICKMODE) .

*** Timed untilstable:
*** First without time constraints:
ceq procParsedTimedCommandTPP('pcheck_|=_untilStable_with`no`time`limit`.,
                              MOD,
                              TERM, BOUND, T, COND,
                              T', COND', SOLVEDTICKMODE) =
      ('\n '\c 'Check '\s '\o eMetaPrettyPrint(MOD, TERM)
      '\c '|= '\o '\s eMetaPrettyPrint(MOD, T) '\y 'untilStable
       '\o '\s
       eMetaPrettyPrint(MOD, T') '\c 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c
       'with 'no 'time 'limit 'with 'mode
       printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
       *** Here comes the real call:
       (if TS :: Term then
          ('\c '\n 'Result: 'the 'property 'does 'not 'hold.
               'Counterexample:  '\n '\o '\t
           eMetaPrettyPrint(MOD, TS) '\o '\n )
        else (if TS == noterm then
                  ('\c '\n 'Result: 'the 'property 'holds. '\o '\n)
              else ('\r '\n 'Error: 'something 'went 'wrong. '\o '\n)
              fi)
        fi)
      )
    if TS := timedUntilStable(pricifyProperties(pricifyMod(MOD)),
                              pricifyInit(getTerm(metaReduce(MOD, TERM))),
                              T,
                              COND, true,
                              T', COND', true, SOLVEDTICKMODE) .

*** Now with time constraints:
ceq procParsedTimedCommandTPPL(Q, MOD,  TERM, BOUND, T, COND,
                               T', COND', LIMIT, SOLVEDTICKMODE) =
      ('\n '\c 'Check '\o '\s eMetaPrettyPrint(MOD, TERM)
      '\c '|= '\o '\s eMetaPrettyPrint(MOD,T) '\y 'untilStable
      '\o '\s
       eMetaPrettyPrint(MOD, T') '\c 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c
       'in 'time commandToCompSymb(Q)
       eMetaPrettyPrint(MOD, LIMIT) '\c
       'with 'mode
       printMode(SOLVEDTICKMODE, MOD) '\c  ': '\n '\o
       *** Here comes the real call:
       (if TS :: Term then
          ('\c '\n 'Result: 'the 'property 'does 'not 'hold.
               'Counterexample: '\n '\o '\t
           eMetaPrettyPrint(MOD, TS) '\o '\n )
        else (if TS == noterm then
                  ('\c '\n 'Result: 'the 'property 'holds. '\o '\n)
              else ('\r '\n 'Error: 'something 'went 'wrong. '\o '\n)
              fi)
        fi)
      )
    if (Q == 'pcheck_|=_untilStable_in`time`<_.) or
       (Q == 'pcheck_|=_untilStable_in`time`<=_.)
       /\ TS := timedUntilStable(pricifyProperties(pricifyMod(MOD)),
                                 pricifyInit(getTerm(metaReduce(MOD, TERM))),
                                 T,
                                 COND, true, T', COND', true,
                                 commandToComp(Q),  LIMIT, SOLVEDTICKMODE) .


*** Model checking! First mc_|=u_. Untimed model checking.
ceq procParsedTimedCommandTP('pmc_|=u_., MOD, TERM, BOUND,
                             T, nil, SOLVEDTICKMODE) =
      if RP :: ResultPair then
       ('\n '\c 'Untimed 'model 'check '\s '\o
                eMetaPrettyPrint(MOD, TERM)
         '\c '\s '|=u '\o '\s eMetaPrettyPrint(MOD, T)
         '\c  '\s 'in '\o
       eMetaPrettyPrint(getName(MOD)) '\c 'with 'mode
          printMode(SOLVEDTICKMODE, MOD) '\c '\n
         '\c '\n 'Result '\o
         eMetaPrettyPrint(getType(RP)) '\c ': '\n '\o '\s '\s
         eMetaPrettyPrint(MOD, getTerm(RP)) '\n '\o )
      else ('\n '\r 'Untimed 'model 'checking: 'Something 'went
                    'wrong! '\o '\n)
      fi
    if RP := metaMC(pricifyProperties(pricifyMod(MOD)),
                    pricifyInit(getTerm(metaReduce(MOD, TERM))),
                    T,
                    'u, SOLVEDTICKMODE) .

ceq procParsedTimedCommandTP('pmc_|=u_., MOD, TERM, BOUND,
                             T, COND, SOLVEDTICKMODE) =
     ('\n '\r 'Error: 'No 'condition 'in
                      'temporal 'logic 'model 'checking!  '\o '\n)
     if COND =/= nil .

*** "Timed", maybe slightly misnamed, model checking.
```

```
  *** Properties valied for GlobalSystems are valid at all times.

  ceq procParsedTimedCommandTP('mc_|=t_with`no`time`limit`., MOD, TERM,
                               BOUND, T, nil, SOLVEDTICKMODE) =
        if RP :: ResultPair then
         ('\n '\c 'Model 'check '\o
                  eMetaPrettyPrint(MOD, TERM) '\s
           '\c '|=t '\o eMetaPrettyPrint(MOD, T)
           '\c '\s 'in '\o
           eMetaPrettyPrint(getName(MOD)) '\c 'with 'mode
            printMode(SOLVEDTICKMODE, MOD) '\c '\n
           '\c '\n 'Result '\o
           eMetaPrettyPrint(getType(RP)) '\c ': '\n '\o '\s '\s
           eMetaPrettyPrint(MOD, getTerm(RP)) '\o '\n)
        else ('\n '\r 'Model 'checking: 'something 'went 'wrong! '\o '\n)
        fi
       if RP := metaMC(pricifyProperties(pricifyMod(MOD)),
                       pricifyInit(getTerm(metaReduce(MOD, TERM))),
                       T,
                      't, SOLVEDTICKMODE) .

  ceq procParsedTimedCommandTP('pmc_|=t_with`no`time`limit`., MOD, TERM, BOUND,
                               T, COND, SOLVEDTICKMODE) =
       ('\n '\r 'Error: 'No 'condition 'in
                        'temporal 'logic 'model 'checking!  '\o '\n)
        if COND =/= nil .

  *** Timed model checking with limit:
  ceq procParsedTimedCommandTPL(Q, MOD, TERM, BOUND, T, nil,
                                LIMIT, SOLVEDTICKMODE) =
        if RP :: ResultPair then
         ('\n '\c 'Model 'check '\o
                  eMetaPrettyPrint(MOD, TERM)
           '\c '\s '|=t '\o eMetaPrettyPrint(MOD, T)
           '\c 'in '\o
         eMetaPrettyPrint(getName(MOD)) '\c '\s
         'in 'time commandToCompSymb(Q)
         eMetaPrettyPrint(MOD, LIMIT) '\c     'with 'mode
            printMode(SOLVEDTICKMODE, MOD) '\c '\n
           '\c '\n 'Result '\o
           eMetaPrettyPrint(getType(RP)) '\c ': '\n '\o '\s '\s
           eMetaPrettyPrint(MOD, getTerm(RP)) '\o '\n)
        else ('\n '\r 'Model 'checking: 'something 'went 'wrong! '\n '\o)
        fi
       if (Q == 'pmc_|=t_in`time`<_.) or (Q == 'pmc_|=t_in`time`<=_.)
          /\ RP := metaMC(pricifyProperties(pricifyMod(MOD)),
                          pricifyInit(getTerm(metaReduce(MOD, TERM))),
                          T,
                          commandToComp(Q), LIMIT, SOLVEDTICKMODE) .

  ceq procParsedTimedCommandTPL(Q, MOD, TERM, BOUND, T, COND,
                                LIMIT, SOLVEDTICKMODE) =
        ('\n '\r 'Error: 'No 'condition 'in
                         'temporal 'logic 'model 'checking!  '\o '\n)
       if (Q == 'pmc_|=t_in`time`<_.) or (Q == 'pmc_|=t_in`time`<=_.)
          /\ COND =/= nil .
endfm

mod PTM-TIMED-DATABASE-HANDLING is
 --- PTM uses these rules to latch onto Real-Time Maude's time ddb handling code
 pr TIMED-DATABASE-HANDLING .

 pr PTM-COMMAND-PROCESSING .
 pr PRICED-UNIT-PROCESSING .

 var ATTS : AttributeSet .
 var DATABASE : DatabaseClass .
 var TIMEDDATABASE : TimedDatabaseClass .
 var DB : Database .
 vars F Q : Qid .
 vars T T' T'' T''' : Term .
 var TL : TermList .
 var O : Oid .
 var MN : ModuleName .
 var TIMEDDATA : TimedData .
 var ME : ModuleExpression .
 var QIL : QidList .


 --- PRICED mods
 crl [databaseToTimedDatabase2] :
     < O : DATABASE | input : (F[T, T']), ATTS >
           =>
           < O : TimedDatabase | input : (F[T, T']), ATTS,
                                 timedData : initTimedData >
        if ((F == 'ptomod_is_endptom)
            or-else (F == 'ptmod_is_endptm)
            or-else (F == 'pomod_is_endpom)
            or-else (F == 'pmod_is_endpm))
           and not (DATABASE :: TimedDatabaseClass) .

  --- PRICED mods
  --- make this new rule or tack on to existing
  crl [readPTimedModule] :
      < O : TIMEDDATABASE | db : DB, input : (F[T, T']),
```

```
                        output : nil, default : ME,
                        timedData : TIMEDDATA, ATTS >
    =>
    < O : TIMEDDATABASE | db : procModule(
                                timedPreModuleToPreModule(
                                    processTimedMetaLevel(F[T, T'],
                                                          TIMEDDATA)),
                                        DB),
                        input : nilTermList,
                        output :
                          ('\n '\c 'Introduced 'timed 'module: '\o
                           header2Qid(parseHeader(T)) '\n),
                        default : parseHeader(T),
                        timedData : addModName(TIMEDDATA,
                                               pureModName(T)),
                        ATTS >
    if  (F == 'ptomod_is_endptom)
    or-else (F == 'ptmod_is_endptm)
    or-else (F == 'pomod_is_endpom)
    or-else (F == 'pmod_is_endpm) .


*** Split the rule TimedExecution to allow for priced execution
*** this separation may be a be artificial but is cleaner
crl [pricedExecution] :
    < O : TIMEDDATABASE | db : DB,
                        input : (F[TL]),
                        output : QIL,
                        default : ME,
                        timedData : TIMEDDATA, ATTS >
    =>
      < O : TIMEDDATABASE | db : DB,
                          input : nilTermList,
                          output : procPriceTimedCommand(F[TL], ME,
                                   DB, getTickMode(TIMEDDATA)),
                          default : ME,
                          timedData : TIMEDDATA, ATTS >
if         --- PRICED stuff
 F == 'binary`find`cheapest_=>*_in`time`<=_. or
 F == 'binary`find`cheapest_=>*_in`time`<_. or
 F == 'binary`find`cheapest_=>*_with`no`time`limit`. or
 F == 'find`cheapest_=>*_in`time`<=_. or
 F == 'find`cheapest_=>*_in`time`<_. or
 F == 'find`cheapest_=>*_with`no`time`limit. or
 F == 'priced`find`earliest_=>*_with`no`cost`limit. or *** add with cost limits
 F == 'priced`find`earliest_=>*_with`cost`<=_. or
 F == 'priced`find`earliest_=>*_with`cost`<_. or
 F == 'ptsearch_=>*_in`time`<_with`cost`<_.  or
 F == 'ptsearch_=>*_in`time`<_with`cost`<=_. or
 F == 'ptsearch_=>*_in`time`<=_with`cost`<_. or
 F == 'ptsearch_=>*_in`time`<=_with`cost`<=_. or
 F == 'ptsearch_=>*_in`time`<_with`cost`<_.  or
 F == 'ptsearch_=>!_in`time`<_with`cost`<=_. or
 F == 'ptsearch_=>!_in`time`<=_with`cost`<_. or
 F == 'ptsearch_=>!_in`time`<=_with`cost`<=_. or
 --- ptsearch with no cost limit not fully implemented yet
 F == 'ptsearch_=>*_in`time`<_with`no`cost`limit.  or
 F == 'ptsearch_=>*_in`time`<_with`no`cost`limit. or
 F == 'ptsearch_=>*_in`time`<=_with`no`cost`limit. or
 F == 'ptsearch_=>*_in`time`<=_with`no`cost`limit. or
 F == 'ptsearch_=>*_in`time`<_with`no`cost`limit.  or
 F == 'ptsearch_=>!_in`time`<_with`no`cost`limit. or
 F == 'ptsearch_=>!_in`time`<=_with`no`cost`limit. or
 F == 'ptsearch_=>!_in`time`<=_with`no`cost`limit. or
 F == 'ptsearch_=>*_with`no`time`limit`with`cost`<_. or
 F == 'ptsearch_=>*_with`no`time`limit`with`cost`<=_. or
 F == 'ptsearch_=>!_with`no`time`limit`with`cost`<_. or
 F == 'ptsearch_=>!_with`no`time`limit`with`cost`<=_. or
 F == 'ptsearch_=>*_with`no`limits`. or
 F == 'ptsearch_=>!_with`no`limits`. or
 F == 'ptfrew_in`time`<=_with`cost`<_. or
 F == 'ptfrew_in`time`<=_with`cost`<=_. or
 F == 'ptfrew_in`time`<_with`cost`<_. or
 F == 'ptfrew_in`time`<_with`cost`<=_. or
 F == 'ptfrew_in`time`<=_with`no`cost`limit. or
 F == 'ptfrew_in`time`<_with`no`cost`limit. or
 F == 'ptfrew_with`no`time`limit`with`cost`<_. or
 F == 'ptfrew_with`no`time`limit`with`cost`<=_. or
 F == 'ptfrew_with`no`limits. or
 F == 'ptrew_in`time`<=_with`cost`<_. or
 F == 'ptrew_in`time`<=_with`cost`<=_. or
 F == 'ptrew_in`time`<_with`cost`<_. or
 F == 'ptrew_in`time`<_with`cost`<=_. or
 F == 'ptrew_in`time`<=_with`no`cost`limit. or
 F == 'ptrew_in`time`<_with`no`cost`limit. or
 F == 'ptrew_with`no`time`limit`with`cost`<_. or
 F == 'ptrew_with`no`time`limit`with`cost`<=_. or
 F == 'ptrew_with`no`limits. or
 *** untimed stuff
 F == 'pfrew_with`cost`<_. or
 F == 'pfrew_with`cost`<=_. or
 F == 'prew_with`cost`<_. or
 F == 'prew_with`cost`<=_. or
 F == 'psearch_=>*_with`cost`<_. or
 F == 'psearch_=>*_with`cost`<=_. or
```

```
   F == 'ut`find`cheapest_=>*_. or
   *** timed model checking
   F == 'pcheck_|=`<>_with`no`time`limit`. or
   F == 'pcheck_|=`<>_in`time`<_. or
   F == 'pcheck_|=`<>_in`time`<=_. or
   F == 'pcheck_|=_until_with`no`time`limit`. or
   F == 'pcheck_|=_until_in`time`<_. or
   F == 'pcheck_|=_until_in`time`<=_. or
   F == 'pcheck_|=_untilStable_with`no`time`limit`. or
   F == 'pcheck_|=_untilStable_in`time`<_. or
   F == 'pcheck_|=_untilStable_in`time`<=_. or
   F == 'pmc_|=u_. or
   F == 'pmc_|=t_with`no`time`limit`. or
   F == 'pmc_|=t_in`time`<_.  or
   F == 'pmc_|=t_in`time`<=_. .
endm
```

In addition some text was appended to Real-Time Maude's banner:

```
mod REAL-TIME-MAUDE is
 ...
  rl [init] :
       init
       => [nil,
           < o : Database |
              db : initialDatabase,
              input : nilTermList, output : nil,
              default : 'CONVERSION >,
           ('\n '\t '\s '\s '\s '\s '\s string2qidList(banner) '\n
           '\n '\t    '\s  '\! '\m  'Real-Time 'Maude '2.2 '\o '\c
               'extension 'October '6 '`, '\s '2006 '\o '\n
               '\n '\t    '\s  '\! '\g  'Priced-Timed 'Maude '1.0 '\o '\c
              'extension 'February '1st '`, '\s '2008 '\o '\n)] .
 ...
endm
```